Open-world Software: Specification, Verification, and Beyond

Doctoral Dissertation submitted to the

Faculty of Informatics of the *Università della Svizzera italiana*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Domenico Bianculli

Dott. Mag. Ing., Politecnico di Milano, Italy

under the supervision of

Prof. Carlo Ghezzi

July 2012

Dissertation Committee

Dissertation accepted on 18 July 2012

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Domenico Bianculli
Lugano, 18 July 2012

To the memory of my mother Serafina
who did not live long enough
to see this accomplishment

In the case of all things which have several parts
and in which the totality is not, as it were, a mere heap,
but the whole is something beside the parts,
there is a cause

ARISTOTLE, *Metaphysics*
(translation by W. D. Ross)

Contents

Figures

Tables

Abstract

Open-world software systems are built by composing heterogeneous, third-party components, whose behavior and interactions cannot be fully controlled or predicted; moreover, the environment they interact with is characterized by frequent, unexpected, and welcome changes. This class of software exhibits new features that often demand for rethinking and extending the traditional methodologies and the accompanying methods and techniques.

In this thesis we deal with a particular class of open-world software, represented by service-based applications (SBAs). We focus on three specific aspects related to the development and provisioning of SBAs: specification, verification, and reputation management. With respect to these aspects, we provide methods and techniques that are i) suitable to deal with aspects such as change, evolution, and reliance on third-parties, and ii) able to improve the overall quality of the systems they are applied to.

More specifically, concerning specification, we report on the findings of a study that analyzed requirements specifications of SBAs developed in research settings and in industrial settings. These findings have then driven the design of SOLOIST, a language used to specify the interactions of SBAs. Regarding verification, our contribution is twofold; we propose: i) a technique for automatically generating the behavioral interfaces of the partner services of a service composition, by decomposing the requirements specification of the composite service; ii) a framework for the definition of verification procedures (encoded as synthesis of semantic attributes associated with a grammar) that are made incremental using an approach based on incremental parsing and attributes evaluation techniques. Finally, as for reputation management, we present a reputation-aware service execution infrastructure, which manages the reputation of services used by composite SBAs in an automated and transparent manner.

Acknowledgments

Pursuing a PhD degree is a long journey; in my case, it took even longer than originally planned. One of the perks of such a long journey was the opportunity to interact with many people, who directly and indirectly contributed to my personal and professional growth, as well as to the end result, i.e., this dissertation. In the following paragraphs I want to say thank you to all of them, with the inherent disclaimer about completeness.

Carlo Ghezzi has been my long-time advisor since 2003. After realizing how much space it would have taken to express my gratitude to him, I decided to manifest it in a more appropriate form elsewhere [34]. Here I want to acknowledge "only" his support and patience, his continuous feedback and constant inspiration, his enthusiasm and friendship.

The official PhD enrollment record at USI lists Mehdi Jazayeri as my *academic advisor*. Any description of this role would be inadequate to explain what Mehdi has represented for me in these six years. Besides keeping track of my progress, he devoted plenty of his time to engage in various conversations with me, during which I learned a lot about research, teaching, academia, university management, the USA, and of course software engineering and programming languages. He gave me the opportunity to be the teaching assistant for his Programming Languages "PL" course: I really enjoyed our discussions on how to shape the course, class by class, edition by edition.

All my co-authors played an important role in developing many ideas presented in this dissertation and provided valuable feedback on my work. Needless to say, they had to become inured with my work schedule and with my constant issues in assigning priority to tasks. Among them, Luciano Baresi, Walter Binder, Dimitra Giannakopoulou, Dino Mandrioli, Corina S. Păsăreanu, Cesare Pautasso, and Pierluigi San Pietro offered me a different, often complementary, perspective of the topics I explored with them. I had also the privilege of working together with some great friends. Paola acted as unofficial mentor during the first years of my PhD, almost continuing her function from the time of my master thesis; over the years, the amount of research work done together has decreased but the intensity of our fabulous friendship has skyrocketed. Sam, the *BPEL master*, has always spared, in each work session, five minutes to check and comment on the latest news (and rumors) about (upcoming) products designed near the *Infinite Loop*. Antonio has been my comrade-in-arms while crossing the broken ground of syntactic-semantic incremental verification during the last months of my

PhD (and in the ones following the graduation, if I can make an educated guess); he is an invaluable associate, not only for his availability for late night conference calls and for putting up with my typography fixation while writing papers.

My PhD studies have been spatially located, over time, in three distinct places that provided a stimulating environment in which to work. USI-INF welcomed me even before I joined as a PhD student and quickly became my new academic home beyond the Swiss-Italian border. I want to thank Alessandra, Alessio, (il) D'Ambros, Regaz, and Tof for the very frequent chats, often on work progress/frustration as well as future, undefined career plans; the members of the REVEAL, STAR, and MJ research groups for the pleasant discussions; Alex Wolf, Michele Lanza, and Antonio Carzaniga for providing—each of them in his specific way—precious advices as PhD program directors; the staff of the Dean's office and the faculty program manager for their administrative support. The DEEP-SE group at Politecnico di Milano represented my second academic home: thanks to all group members and in particular to all fellow PhD students for the many interesting discussions (definitely not always on scientific matters!) as well as the amusing group lunches/dinners/parties. Dimitra Giannakopoulou and Corina S. Păsăreanu kindly agreed to host me at NASA Ames Research Center; special thanks go to the two of them for trusting me even in the darkest moments of dashed hope. I also want to thank Christina and Saba of Mission Critical Technologies Inc. for their administrative support; my Summer 2009 fellow interns (especially those wearing a glaring red, restricted access badge), the members of the Robust Software Engineering group, and the other MCT employees on site at Ames for the many lunches and dinners together; Franco Raimondi for bringing a touch of Italy during that Californian summer and for sharing his tips and his experience as a postdoc-in-London with me.

Part of the work presented in chapter 3 was performed thanks to the kind availability of Credit Suisse AG, specifically Mr. Patrick Senti and his team.

During the six years of my PhD I also had the pleasure of meeting and interacting with many colleagues at the annual conference(s) on software engineering, and also as part of the PLASTIC and S-CUBE research projects.

I was very lucky to have some amazing friends that helped me in having a life outside the academic world. Thanks to the many friends (based) in Moliterno, Milano, Roma, and the Santa Clara county, who were always there for a chat, a call, *un aperitivo*, a drink or a dinner together, and also for hosting me at their place during my quick trips. Special thanks go to the set of "special" friends that I like to call the *stars*: {*Gomeisa*, *Gienah*, *Spica*, *Vega*, *Porrima*, *Nekkar*, *Markab*, *Furud*, {*Formalhaut*, *Girtab*}, *Acrab*, *Meissa*, {*Alpheratz*, *Caph*}}. Some of them know the meaning behind this associative array, some will eventually know, some will probably never know; in any case, they brought, often unawares, *light* in my moments of darkness.

Finally, I want to express my gratitude to mom (RIP), dad, and all my relatives, for being supportive and always understanding me and my geographical distance.

Part I

Overture

Chapter 1

Introduction

1.1 Open-world Software

Early approaches to software engineering, developed in the late ’60s and in the early ’70, were targeted to discipline the software process and improve software quality, by defining exact stages and proposing criteria to step from one stage to another. For example, one of these early attempts is represented by the waterfall development process [130].

These early approaches were proposed based on some fundamental assumptions, which reflected the way software was developed and meant to operate at that time:

- organizations were *monolithic*;
- software development was *centralized* inside each organization;
- software modules were *statically* bound to each other;
- the final system was deployed on a *well-known infrastructure*, which could be even *physically centralized*;
- the environment with which a software interacted (i.e., the world) was assumed to be *static,* that is software requirements were considered to be *stable*.

All these assumptions represent what is collectively called *closed-world assumption* in [11].

Nevertheless, since the early proposal of Parnas to “design for change” [121], in the last three decades software engineering has shifted towards a type of software that is characterized by a different set of assumptions:

- software development and provisioning is *decentralized*, since it involves *multiple stakeholders* belonging to *different organizations*;

- systems are assembled out of *components* that provide a specific functionality and are provided by *independent third-parties*;
- bindings among components are often *delayed until the execution*;
- bindings among components may *dynamically vary* to accommodate changes that support the *evolution* of the environment with which the software system interacts;
- physical deployment of the system requires a *heterogeneous and distributed network infrastructure*.

In their seminal paper [11], Baresi et al. collectively call these assumptions *open-world assumption*; the software developed following these assumptions is consequently called *open-world software*.

In the last years, service-oriented architectures (SOAs [82]) have emerged as a promising solution to the problem of developing decentralized, distributed, and evolvable applications [55]. In these architectures, *services* represent software components that provide specific functionality, exposed for possible use by many clients, who can dynamically discover and access them through network infrastructures. This architectural paradigm has been adopted in new and innovative computing domains, like ambient intelligence, context-aware applications, and pervasive computing. Many technologies, such as Web services, Jini and OSGi, have been associated with SOAs.

The applications developed according to the principles of service orientation are called *service-based applications* (SBAs[1]). SBAs are developed, deployed, and operated by organizations that behave as *service providers*, and they are used by different *client* organizations. Clients can be final users or they can act as *service integrators*, who provide new added-value services by composing existing services, possibly offered by others. Service compositions, also called service orchestrations, can be defined in high-level, workflow-style languages such as BPEL [5].

SBAs represent an instantiation of the class of open-world software. Indeed, one peculiarity of this class of software is their intrinsic tendency to change and evolve, dynamically and autonomously [120]. A simple example of such change is represented by a provider performing a regular maintenance activity, which could modify an existing service into an upgraded but, regrettably, incorrect and/or incompatible version, which could break the compatibility or the service level agreement (SLA) with its existing clients. In some cases, a service provider could dynamically modify the exported service in a malicious way, for example offering a lower-quality service than the one promised through the SLA. Furthermore, new services may be developed and published in registries, and then discovered dynamically by possible clients; conversely, previously available services may disappear or become unavailable. Moreover, service

[1]In the rest of this document, we use the terms “SBA” and “service” interchangeably.

compositions may make use of dynamic binding techniques to support continuous evolution and contextual adaptation. This implies that at design time the external services orchestrated in a service composition are only known through their abstract interface, while their concrete identity may become known only later at run time, when bindings are resolved.

Since an SBA consists of a composition of existing services, no complete view and control of the entire application is in the hands of a single organization, but rather a multi-stakeholder playground emerges, which requires new strategies to deal with the decentralized and autonomous evolution of different parts of a system.

1.2 Problem Statement and Research Goals

In [11], the authors defined a research agenda for open-world software, identifying new challenges related to specification, verification[2], monitoring, trust, implementation, and self-management. In this thesis we decided to take up three of them, contextualized in the domain of SBAs, as described in the formulation of the problem statement:

> *Open, dynamic software systems, such as applications built out of the composition of services, demand rethinking methods and techniques for **specification**, **verification**, and **reputation**[3] **management**, to cope with the specific facets of this class of software.*

This statement leads to the definition of the following overall research goal:

> *To design new methods and techniques for specification, verification, and reputation management of open-world software, in particular for the case of service-based applications. These methods and techniques should be i) suitable to deal with aspects such as change, evolution, and reliance on third-parties, and ii) able to improve the overall quality of these applications.*

Research Goals

The overall research goal can be decomposed into four smaller research goals, described below.

[2]In this thesis, the term *verification* is used in a general and broad sense, which encompasses all the activities undertaken to ascertain that a software meets its objectives. Often the term *validation* [40] is also used vis-a-vis verification to indicate specific activities (and goals) and *V&V* (verification and validation) is used as a catch-all term.

[3]We assume *reputation* as a basic concept associated with *trust*.

Research goal 1 - Specification language

Specifying the interactions of a service composition with its partner services encompasses different functional and non-functional aspects, which might not be completely supported by traditional specification languages. This calls for new specification languages, tailored for the domain of SBAs, as expressed by the first research goal:

> ***(RG1)*** *To understand the expressiveness requirements of a specification language that aims at describing both functional and non-functional properties of the interactions of a service composition with its partner services, and to develop a specification language based on these requirements.*

Research goal 2 - Change-aware verification

Given the dynamic and evolving nature of SBAs, change management practices applied in the context of SBAs impose time constraints that are often too costly, in terms of execution time and memory consumption, for verification techniques. Hence, the second research goal can be formulated as:

> ***(RG2)*** *To develop verification techniques that can deal efficiently with changes occurring in SBAs.*

Research goal 3 - Generation of the interface specification of third-party services

The external services with which a service composition interacts are usually known by means of their syntactical interface. However, an interface providing more information, such as a behavioral specification, could be more useful to a service integrator for assessing that a certain external service can contribute to fulfill the functional requirements of the composite application. Thus, the third research goal is:

> ***(RG3)*** *To design an analysis technique to generate the behavioral interfaces of the external services, given the requirements specification of a composite service.*

Research goal 4 - Reputation management

The overall correctness and quality of service of composite services is largely affected by their constituent web services. Composite services have to operate in an open and dynamically changing environment in order to leverage the best performing services available at the moment. Hence, there is the need for an efficient mechanism to provide reliable service rankings and to exploit them at run time. Accordingly, the fourth research goal is:

> ***(RG4)*** *To design a run-time infrastructure that manages services' reputation and exploit it to enable self-tuning and self-healing properties in the execution of composite services.*

1.3 Contributions

In this section we outline the contributions of the thesis, mapping them to the research goals stated above.

Research goal 1 - Specification language

The contributions addressing this research goal are as follows:

Analysis of property specification patterns in SBAs. We performed a study on the use of property specification patterns in SBAs, by comparing the usage of specification patterns in published research case studies—representing almost ten years of research in the area of specification, verification, and validation of SBAs—with a large body of specifications written by our industrial partner over a similar time period. The outcome of this study indicated new requirements for the development of specification languages for SBAs. This study is described in chapter 3.

The SOLOIST specification language. Based on the results of the aforementioned study, we designed from scratch SOLOIST, a specification language for service composition interactions, driven by the requirements of expressiveness and support for automated verification tools. SOLOIST is illustrated in chapter 4.

Research goal 2 - Change-aware verification

The contribution addressing this research goal is the following:

A syntactic-semantic approach for incremental verification. We designed SiDECAR, a general framework for the definition of verification procedures, which are made incremental by the framework itself. The analysis executed within the verification procedure is driven by the syntactic structure of the software system. The verification procedure is encoded within the semantic attributes associated with the grammar generating the system description. Incrementality is achieved by coupling the evaluation of semantic attributes with an incremental parsing technique. The framework enables the definition and the execution of efficient, incremental verification procedures. SiDECAR is illustrated in chapter 7.

Research goal 3 - Generation of the interface specification of third-party services

This research goal has been addressed with the following contribution:

Interface decomposition for service compositions. We developed a technique for automatically generating the behavioral interfaces of the partner services of a service compositions, by *decomposing* the requirements specification of a service composition. The technique generates behavioral interfaces that constitute required specifications for the partner services; these specifications guarantee that the composite service will fulfill its required safety properties at run time, while it interacts with the external services. Since we assume that the behavioral descriptions of external services are not available, our technique is based on the purely syntactical knowledge of their interfaces. This technique is presented in chapter 6.

Research goal 4 - Reputation management

This research goal has been addressed with the following contribution:

A pro-active reputation management infrastructure for composite Web services. We designed ReMan, a reputation management infrastructure for composite Web services. It supports the aggregation of clients' feedback on the perceived quality of service (QoS) of external services, using reputation mechanisms to build service rankings. Changes in rankings are pro-actively notified to composite service clients to enable self-tuning properties in their execution. ReMan is described in chapter 9.

1.4 Dissemination

The research work we performed during the PhD program has lead to several publications. This section lists them, divided into two categories: i) publications that are fundamental for the thesis contributions; and ii) publications that are related to the thesis.

Conference papers

- D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. Specification patterns from research to industry: a case study in service-based applications. In *ICSE 2012: Proceedings of the 34th International Conference on Software Engineering*, pages 968–976. IEEE, 2012.

  This paper is the basis of chapter 3. It presents the results of our study—performed in collaboration with Credit Suisse AG—on specification patterns for service-based applications, focused on industrial SBAs in the banking domain. The outcome of this study deeply influenced the design of the SOLOIST language, introduced in chapter 4.

- D. Bianculli, C. Ghezzi, and P. San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *FACS 2012: Proceedings of the*

9th International Symposium on Formal Aspects of Component Software, 2012. To appear.

This paper is the basis of chapter 4. It contains the definition of the SOLOIST language as well as its translation into linear temporal logic.

- D. Bianculli, D. Giannakopoulou, and C. S. Păsăreanu. Interface decomposition for service compositions. In *ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering*, pages 501–510. ACM, 2011.

  This paper is the basis for chapter 6. It illustrates a technique for decomposing interface specifications of service compositions. This work has been developed as part of our collaboration with the Robust Software Engineering group of NASA Ames Research Center.

- D. Bianculli, W. Binder, L. Drago, and C. Ghezzi. Transparent reputation management for composite Web services. In *ICWS 2008: Proceedings of the IEEE International Conference on Web Services*, pages 621–628. IEEE, 2008,

  and

- D. Bianculli, W. Binder, L. Drago, and C. Ghezzi. ReMan: A pro-active reputation management infrastructure for composite Web services. In *ICSE 2009: Proceedings of the 31st International Conference on Software Engineering*, pages 623–626. IEEE, 2009. Formal Research Demo.

  These two papers are the basis of chapter 9. They present a reputation infrastructure to automatically and transparently monitor the execution of composite services. The infrastructure enables self-tuning and self-healing properties in the execution of composite services.

Unpublished reports

- D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. A syntactic-semantic approach to incremental verification. Internal Report.

  Part of this report is the basis for chapter 7. The report presents SiDECAR and its application to define two kinds of verification: software reliability analysis and reachability analysis.

Related publications

Journal papers

- L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, 2007.

Book chapters

- D. Bianculli, C. Ghezzi, P. Spoletini, L. Baresi, and S. Guinea. A guided tour through SAVVY-WS: a methodology for specifying and validating Web service compositions. In *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 131–160. Springer, 2008.

Conference papers

- D. Bianculli, W. Binder, and M. L. Drago. Automated performance assessment for service-oriented middleware: a case study on BPEL engines. In *WWW 2010: Proceedings of the 19th International Conference on World Wide Web*, pages 141–150. ACM, 2010.

- D. Bianculli, W. Binder, and M. L. Drago. SOABench: Performance evaluation of service-oriented middleware made easy. In *ICSE 2010: Proceedings (Volume 2) of the 32nd International Conference on Software Engineering*, pages 301–302. ACM, 2010. Informal Research Demo.

- L. Baresi, D. Bianculli, S. Guinea, and P. Spoletini. Keep it small, keep it real: Efficient run-time verification of web service compositions. In *FMOODS/FORTE 2009: Proceedings of IFIP international conference on Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 26–40. Springer, 2009.

- D. Bianculli, R. Jurca, W. Binder, C. Ghezzi, and B. Faltings. Automated dynamic maintenance of composite services based on service reputation. In *ICSOC'07: Proceedings of the 5th International Conference on Service-oriented computing*, volume 4749 of *LNCS*, pages 449–455. Springer, 2007.

- L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. A timed extension of WSCoL. In *ICWS 2007: Proceedings of the IEEE International Conference on Web Services*, pages 663–670. IEEE, 2007.

- D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify BPEL4WS workflows. In *SOCA 2007: Proceedings of the 2007 IEEE International Conference on Service-Oriented Computing and Applications*, pages 13–20. IEEE, 2007.

- D. Bianculli, A. Morzenti, M. Pradella, and P. San Pietro and Paola Spoletini. Trio2Promela: a model checker for temporal metric specifications. In *ICSE 2007 Companion: Companion of the proceedings of the 29th International Conference on Software Engineering*, pages 61–62. IEEE, 2007. Informal Research Demo.

- D. Bianculli, P. Spoletini, A. Morzenti, M. Pradella, and P. San Pietro. Model checking temporal metric specification with Trio2Promela. In *FSEN 2007: Proceedings of International Symposium on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 388–395. Springer, 2007.

Workshop papers

- D. Bianculli, C. Ghezzi, and C. Pautasso. Embedding continuous lifelong verification in service life cycles. In *PESOS 2009: Proceedings of the First International Workshop on Principles of Engineering Service-oriented Systems*, pages 99–102. IEEE, 2009.
- D. Bianculli. Lifelong verification of dynamic service compositions. In *FSEDS '08: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium, co-located with ACM SIGSOFT 2008/FSE 16*, pages 1–4. ACM, 2008.
- D. Bianculli and C. Ghezzi. SAVVY-WS at a glance: supporting verifiable dynamic service compositions. In *ARAMIS 2008: Proceedings of the 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems*, pages 49–56. IEEE, 2008.
- D. Bianculli and C. Ghezzi. Towards a methodology for lifelong validation of service compositions. In *SDSOA 2008: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments*, pages 7–12. ACM, 2008.
- D. Bianculli and C. Ghezzi. Monitoring conversational web services. In *IW-SOSWE'07: Proceedings of the 2nd International Workshop on Service-Oriented Software Engineering*, pages 15–21. ACM, 2007.

1.5 Structure of the Thesis

The rest of this thesis is structured as follows. Chapter 2 provides background information on service compositions and some formal models used in the following chapters. There are then three parts, each one corresponding to one of the main challenges related to open-world software tackled in this thesis. Part II, on specification, includes chapter 3 on the study of property specification patterns, chapter 4 on the SOLOIST language, and chapter 5, which provides a short summary of the relevant literature related to the content of the part. Part III contains chapter 6 on the technique for decomposing interface specifications of service compositions, chapter 7 on SiDECAR and incremental verification, and chapter 8, which summarizes the state of the art relevant for this part. Part IV, with chapter 9, describes the ReMan reputation management infrastructure. Finally, chapter 10 concludes the thesis, by discussing open issues and future research directions.

Chapter 2

Background

This chapter provides some background information on notations and formal models used in the remaining of the thesis. Section 2.1 introduces BPEL [5], the de facto standard for defining composite applications based on Web services. All the examples of service compositions presented in this thesis are defined in BPEL. Section 2.2 presents concepts related to temporal logics, which are then used in chapter 4. Section 2.3 formally defines labeled transition systems (LTSs) and the operations that can be performed over them; LTSs are used in chapter 6 to model (and specify) the service behavior. Section 2.4 introduces Floyd grammars and attribute grammars, at the base of the approach described in chapter 7.

2.1 BPEL in a Nutshell

BPEL —Business Process Execution Language (for Web Services)—is a high-level XML-based language for the definition and execution of business processes. It supports the definition of workflows that provide new services, by composing external Web services in an orchestrated manner. The definition of a workflow contains a set of global variables and the workflow logic is expressed as a composition of *activities*; variables and activities can be defined at different visibility levels within the process using the *scope* construct.

Activities include primitives for communicating with other services (*receive*, *invoke*, *reply*), for executing assignments (*assign*) to variables, for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of a process (*terminate*). Moreover, constructs like *sequence, while,* and *switch* provide standard control structures to order activities and to define loops and branches. The *pick* construct makes the process wait for the arrival of one of several possible incoming messages or for the occurrence of a time-out, after which it executes the activities associated with the event.

The language also supports the concurrent execution of activities by means of the *flow* construct. Synchronization among the activities of a *flow* may be expressed using

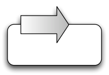
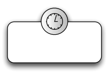
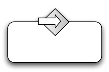
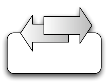
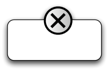
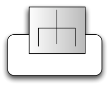
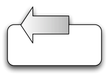
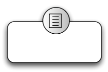
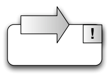
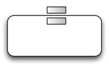
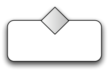
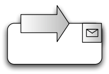
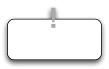
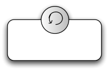
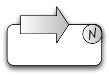
Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

Figure 2.1. **Graphical notation for BPEL**

the *link* construct; a link can have a guard, which is called *transitionCondition*. Since an activity can be the target of more than one link, it may define a *joinCondition* for evaluating the *transitionCondition* of each incoming link. By default, if the *joinCondition* of an activity evaluates to false, a fault is generated. Alternatively, BPEL supports *Dead Path Elimination*, to propagate a false condition rather than a fault over a path, thus disabling the activities along that path.

Each *scope* (including the top-level one) may contain the definition of the following handlers:

- An *event handler* reacts to an event by executing—concurrently with the main activity of the *scope*—the activity specified in its body. In BPEL there are two types of events: message events, associated with incoming messages, and alarms based on a timer.

- A *fault handler* catches faults in the local *scope*. If a suitable *fault handler* is not defined, the fault is propagated to the enclosing *scope*.

- A *compensation handler* restores the effects of a previously completed transaction. The *compensation handler* for a *scope* is invoked by using the *compensate* activity, from a *fault handler* or *compensation handler* associated with the parent *scope*.

The graphical notation for BPEL activities used across this thesis is shown in figure 2.1.

2.2 Temporal Logics

The language of PLTL, Linear Temporal Logic with past operators [86], is composed by the following elements:

1. a set Π of *atomic proposition*;

2. two *propositional connectives*, $\neg$, $\wedge$ (from which the other traditional connectives can be defined);

3. four *temporal operators*: X ("next"), Y ("yesterday"), U ("until"), S ("since").

The syntax of PLTL formulae is given by the following rules:

- if $\phi \in \Pi$, then ϕ is a formula;

- if ϕ and ψ are formulae, then $\neg\phi$, $\phi \wedge \psi$, $\phi \mathsf{U} \psi$, $\phi \mathsf{S} \psi$, $\mathsf{X}\phi$, $\mathsf{Y}\phi$ are formulae;

- nothing else is a formula.

Other temporal operators may be defined from the primitive ones; for example, the "eventually" operator can be defined as $\mathsf{F}\phi \equiv \top \mathsf{U} \phi$

The semantics of PLTL is defined on ω-words. Given a finite alphabet Σ, an ω-word over Σ is an infinite sequence $w = w_0 w_1 w_2 \ldots$, with $w_i \in \Sigma$ for every $i \geq 0$. An element w_i of $w = w_0 w_1 w_2 \ldots$ is denoted as $w(i)$. For all PLTL formulae ϕ, for all $w \in \left(2^{\Pi}\right)^{\omega}$, for all natural numbers i, the satisfaction relation $w, i \models \phi$ is defined as follows:

$w, i \models p$	iff	$p \in w(i)$, with $p \in \Pi$
$w, i \models \neg\phi$	iff	$w, i \not\models \phi$
$w, i \models \phi \wedge \psi$	iff	$w, i \models \phi$ and $w, i \models \psi$
$w, i \models \mathsf{X}\phi$	iff	$w, i+1 \models \phi$
$w, i \models \phi \mathsf{U} \psi$	iff	for some $k > 0$: $w, i+k \models \psi$,
		and for all $j \, 0 < j < k$: $w, i+j \models \phi$
$w, i \models \mathsf{Y}\phi$	iff	$i > 0$ and $w, i-1 \models \phi$
$w, i \models \phi \mathsf{S} \psi$	iff	for some $k > 0$: $i - k \geq 0, w, i-k \models \psi$,
		and for all $j \, 0 < j < k$: $w, i-j \models \phi$

A PLTL formula ϕ is satisfied on an ω-word w iff $w, 0 \models \phi$.

2.3 Labeled Transition Systems

Labeled Transition Systems are defined as follows. Let *Act* be the universal set of observable actions and let τ denote an internal action that cannot be observed by the environment of a component. Let π denote a special *error state*, which models safety violations in the associated transition system. A *Labeled Transition System M* is a 4-tuple $\langle Q, A, \delta, q_0 \rangle$ where Q is a finite non-empty set of states; $A = \alpha M \cup \{\tau\}$, with $\alpha M \subseteq$

Act is the actions alphabet; $\delta \subseteq Q \times A \times Q$ is a transition relation; $q_0 \in Q$ is the initial state. Moreover, let Π denote a special LTS defined as $\Pi = \langle \{\pi\}, Act, \varnothing, \pi \rangle$.

An LTS $M = \langle Q, A, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ-transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

An LTS is *complete* if in each state a transition is defined upon each action of the alphabet; more formally, $M = \langle Q, \alpha M \cup \{\tau\}, \delta, q_0 \rangle$ is complete if and only if $\forall q \in Q, \forall a \in \alpha M, \exists q' \in Q \mid (q, a, q') \in \delta$. If an LTS M is not complete, it can be completed with a sink state and the transitions leading to it; the resulting LTS is denoted as $\hat{M}$. Formally, given an LTS $M = \langle Q, \alpha M \cup \{\tau\}, \delta, q_0 \rangle$, its complete-by-construction variant is $\hat{M} = \langle Q \cup \{\hat{q}\}, \alpha\hat{M} \cup \{\tau\}, \delta', q_0 \rangle$, where $\alpha\hat{M} = \alpha M, \delta' = \delta \cup \{(\hat{q}, a, \hat{q}) \mid a \in \alpha M\} \cup \{(q, a, \hat{q}) \mid a \in \alpha M \wedge \neg\exists q' \in Q \mid (q, a, q') \in \delta\}$.

For an LTS $M = \langle Q, A, \delta, q_0 \rangle$, there is a *path* σ from state q to state q', with $q, q' \in Q$, if there exists a set of states $\{q_1, \ldots, q_n\} \subseteq Q$ and a sequence of actions $\langle a_1, \ldots, a_{n-1} \rangle$, with each $a_i \in A$, such that $q = q_1 \wedge q' = q_n \wedge \forall i, 1 \leq i \leq n-1, (q_i, a_i, q_{i+1}) \in \delta$. The sequence of actions $\langle a_1, \ldots, a_{n-1} \rangle$, where the τ-transitions are ignored, is called the trace defined by the path σ. A *trace* of an LTS M is a trace defined by a path that originates in the initial state; i.e., it is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state. The set of traces of M is denoted as $Tr(M)$. For an LTS M, $errTr(M) \subseteq Tr(M)$ is the set of traces $\{t \in Tr(M) \mid \exists$ a path σ from q_0 to π and t is defined by $\sigma\}$; $errTr(M)$ is called the set of *error traces* of M. Furthermore, given a trace t and a set $\mathscr{A} \subseteq Act$, the expression $(t \upharpoonright \mathscr{A})$ denotes the trace obtained from t by removing all occurrences of actions $a \notin \mathscr{A}$; "$\upharpoonright$" is the *restriction* operator for traces.

In some cases, it might be useful to explicitly indicate that an LTS has the error state π, reachable from the initial state. For an LTS $M = \langle Q, A, \delta, q_0 \rangle$, we use the notation M_π if and only if $\pi \in Q$ and $errTr(M) \neq \varnothing$. This notation can be combined with the one denoting the completion-by-construction, as in $\hat{M}_\pi$, to identify an LTS that is complete and that contains the error state (reachable from the initial state).

Operators

Let $M = \langle Q, A, \delta, q_0 \rangle$ and $M' = \langle Q', A', \delta', q'_0 \rangle$, with $q'_0 \neq \pi$. M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if $(q_0, a, q'_0) \in \delta$, with $Q = Q', A = A', \delta = \delta'$. Moreover, we say that M *transits* into Π with action a, denoted as $M \xrightarrow{a} \Pi$, if $(q_0, a, \pi) \in \delta$.

The *interface* operator "$\uparrow$" is used to make unobservable some actions of an LTS. Given an LTS $M = \langle Q, A, \delta, q_0 \rangle$ and a set of observable actions $\mathscr{A} \subseteq Act$, $M \uparrow \mathscr{A}$ is defined as follows. If $M = \Pi, M \uparrow \mathscr{A} = \Pi$. For $M \neq \Pi, M \uparrow \mathscr{A} = \langle Q, (\alpha M \cap \mathscr{A}) \cup \{\tau\}, q_0, \delta' \rangle$, where δ' is described by the rules shown in figure 2.2a. The semantics of this operator ensures that $errTr(M) \neq \varnothing$ if and only if $errTr(M \uparrow \mathscr{A}) \neq \varnothing$.

Two LTSs can be combined by means of the *parallel composition* "$\|$" operator, which is commutative and associative. Given two LTSs $M_1 = \langle Q_1, A_1, \delta_1, q_0^1 \rangle$ and $M_2 =$

$$\frac{M \xrightarrow{a} M', a \in \mathscr{A}}{M \uparrow \mathscr{A} \xrightarrow{a} M' \uparrow \mathscr{A}} \qquad \frac{M \xrightarrow{a} M', a \notin \mathscr{A}}{M \uparrow \mathscr{A} \xrightarrow{\tau} M' \uparrow \mathscr{A}}$$

(a) **Rules for the interface operator**

$$\frac{M_1 \xrightarrow{a} M_1'}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2} \quad \frac{M_2 \xrightarrow{a} M_2'}{M_1 \parallel M_2 \xrightarrow{a} M_1 \parallel M_2'} \quad \frac{M_1 \xrightarrow{a} M_1', M_2 \xrightarrow{a} M_2'}{M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'}$$

$$a \notin \alpha M_2 \qquad\qquad a \notin \alpha M_1 \qquad\qquad a \in (\alpha M_1 \cap \alpha M_2)$$

(b) **Rules for the parallel composition operator**

Figure 2.2. **Rules for the LTS operators**

$\left\langle Q_2, A_2, \delta_2, q_0^2 \right\rangle$, the parallel composition $M_1 \parallel M_2$ is defined as follows. If either $M_1 = \Pi$ or $M_2 = \Pi$, then $M_1 \parallel M_2 = \Pi$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, A, \delta, q_0 \rangle$ where $Q = Q_1 \times Q_2, q_0 = (q_0^1, q_0^2), A = A_1 \cup A_2$ and δ is described by the rules shown in figure 2.2b.

The traces of a parallel composition are defined as follows: $Tr(M_1 \parallel M_2) = \{t \mid (t \upharpoonright \alpha M_1) \in Tr(M_1) \wedge (t \upharpoonright \alpha M_2) \in Tr(M_2) \wedge t \in (\alpha M_1 \cup \alpha M_2)^*\}$. As for error traces, a parallel composition has an error trace if at least one of its components has an error trace. In symbols: $errTr(M_1 \parallel M_2) = \{t \in Tr(M_1 \parallel M_2) \mid (t \upharpoonright \alpha M_1) \in errTr(M_1) \vee (t \upharpoonright \alpha M_2) \in errTr(M_2)\}$.

Safety Properties

A safety property can be specified as a deterministic LTS that contains no π state. The set of traces $Tr(P)$ of a property P defines the set of acceptable behaviors over αP. An LTS M satisfies P, denoted as $M \models P$ if and only if $Tr(M \uparrow \alpha P) \subseteq Tr(P)$. For a property LTS P we can define the *error LTS* P_{err} as follows: given $P = \langle Q, \alpha P, \delta, q_0 \rangle$, $P_{err} = \langle Q \cup \{\pi\}, \alpha P_{err}, \delta', q_0 \rangle$, where $\alpha P_{err} = \alpha P$, $\delta' = \delta \cup \{(q, a, \pi) \mid (q, a) \in Q \times \alpha P \wedge \neg \exists q' \in Q \mid (q, a, q') \in \delta\}$. Note that the error LTS is complete by construction[1].

Let M be an LTS such that $errTr(M) = \emptyset$. We detect possible violations of a property P by the component M by computing $M \parallel P_{err}$. As shown in [46], the execution of M leads to a violation of a property P if and only if $errTr(M \parallel P_{err}) \neq \emptyset$, i.e., if and only if the π state is reachable in $M \parallel P_{err}$.

[1]Since an error LTS models a safety property violation, it is customary not to include self-loops for π, which are implied.

2.4 Floyd Grammars and Attribute Grammars

The definitions provided in this section are based on [50]. For more information on formal languages and grammars, we refer the reader to classic textbooks such as [132].

Floyd Grammars

A *context-free (CF)* grammar G is a tuple $G = \langle V_N, V_T, P, S \rangle$, where V_N is a finite set of non-terminal symbols; V_T is a finite set of terminal symbols, disjoint from V_N; $P \subseteq V_N \times (V_N \cup V_T)^*$ is a relation whose elements represent the productions of the grammar; $S \in V_N$ is the axiom or start symbol. We use the following naming convention, unless otherwise specified: lowercase letters at the beginning of the alphabet (a, b, c) denote terminal symbols, while those at the end of the alphabet (u, v, x, y, w, z) denote terminal strings; symbols enclosed in chevrons, such as $\langle A \rangle$, denote non-terminal symbols.

For a grammar G, the *immediate derivation* relation, denoted by $\Rightarrow$, is defined on $(V_N \cup V_T)^*$: $\alpha \Rightarrow \beta$ if and only if $(\gamma, \delta) \in P$, $\gamma \in V_N$, $\alpha_1, \alpha_2, \beta \in (V_N \cup V_T)^*$, such that $\alpha = \alpha_1 \gamma \alpha_2$ and $\beta = \alpha_1 \delta \alpha_2$. Its reflexive transitive closure is denoted by $\overset{*}{\Rightarrow}$.

A production is in *operator form* if its right hand side has no adjacent non-terminals, and an *operator grammar* contains only productions in operator form. Any CF grammar admits an equivalent operator grammar [132]. A classic example of an operator grammar is the one generating arithmetic expressions, shown in figure 2.3a, where '**n**' stands for any number.

Floyd grammars [63], also called operator precedence grammars[2], can be defined starting from operator grammars by effectively defining a binary relation on V_T named *precedence*. Given two terminals, the precedence relation between them can be one of three types: *equal-precedence* ($\doteq$), *takes-precedence* ($\gtrdot$), and *yields-precedence* ($\lessdot$). The meaning of the precedence relation is analogous to the one between arithmetic operators. It can be computed in a fully automatic way for any operator grammar; see [61] for more details. It is convenient to represent the precedence relation in a $V_T \times V_T$ matrix, named *operator precedence matrix (OPM)*. An entry $m_{a,b}$ of an OPM represents the set of operator precedence relations holding between terminals a and b. For example, figure 2.3b shows the OPM for the grammar depicted in figure 2.3a. Precedence relations have not to be total, nor symmetric. If an entry $m_{a,b}$ of an OPM M is empty, the occurrence of the terminal a followed by the terminal b represents a malformed input, which cannot be generated by the grammar.

Definition 1 (Floyd Grammars)**.** *G is a Floyd grammar if and only if its OPM is a conflict-free matrix, i.e., for each* $a, b \in V_T, |m_{a,b}| \leq 1$.

[2]We follow the convention introduced in [50], which names this kind of grammars *Floyd grammars* to honor the memory of Robert Floyd and also to avoid confusion with other similarly named but quite different types of precedence grammars.

$$\begin{array}{l} \langle E \rangle ::= \langle E \rangle \text{ `}\mathbf{+}\text{'} \langle T \rangle \mid \langle T \rangle \\ \langle T \rangle ::= \langle T \rangle \text{ `}\mathbf{*}\text{' `}\mathbf{n}\text{'} \mid \text{`}\mathbf{n}\text{'} \end{array}$$

(a)

$$\begin{array}{c|ccc} & \text{`}\mathbf{n}\text{'} & \text{`}*\text{'} & \text{`}\mathbf{+}\text{'} \\ \hline \text{`}\mathbf{n}\text{'} & & \gtrdot & \gtrdot \\ \text{`}*\text{'} & \doteq & & \\ \text{`}\mathbf{+}\text{'} & \lessdot & \lessdot & \gtrdot \end{array}$$

(b)

Figure 2.3. **Example of an operator grammar and its operator precedence matrix**

Attribute Grammars

Attribute Grammars have been proposed by Knuth as a way to express the semantics of programming languages [89]. Attribute grammars extend CF grammars by associating *attributes* and semantic functions to the production rules of a CF grammar; attributes define the "meaning" of the corresponding nodes in the syntax tree of a sentence generated by the grammar.

Attributes can be of two types: *synthesized* attributes characterize an information flow in bottom-up direction, from descendent nodes (of a syntax tree) to their ancestors; *inherited* attributes are used to specify the flow of information top-down, from ancestors to descendants. In this thesis we consider only synthesized attributes. In fact, it can be proved that they are semantically complete, meaning that any inherited attribute can be translated into a (set of) synthesized attribute(s) reproducing the same information [89].

An attribute grammar that has only synthesized attributes is called an *S-attributed grammar*. Such a grammar can be obtained from a context-free grammar G by adding a finite set of attributes SYN and a set SF of semantic functions. Each symbol $\langle X \rangle \in V_N$ has a set of (synthesized) attributes $SYN(\langle X \rangle)$; $SYN = \bigcup_{\langle X \rangle \in V_N} SYN(\langle X \rangle)$. We use the symbol α to denote a generic element of SYN; we assume that each α takes values from a corresponding domain T_α. The set SF consists of functions, each of them associated with a production in P. For each attribute α of the left hand side of p, a function $f_{p\alpha} \in SF$ synthesizes the value of α based on (a subset of) the attributes of the elements constituting the right hand side of p.

For example, the grammar in figure 2.3a can be extended to an attribute grammar that computes the value of an expression. All nodes have only one attribute called *value*, with $T_{value} = \mathbb{N}$. The set of semantic functions SF can be directly defined as below, where semantic functions are enclosed in braces next to each production:

$$\begin{array}{lll} \langle E_0 \rangle ::= \langle E_1 \rangle \text{ `}\mathbf{+}\text{' } \langle T \rangle & \{value(\langle E_0 \rangle) & = value(\langle E_1 \rangle) + value(\langle T \rangle)\} \\ \langle E \rangle ::= \langle T \rangle & \{value(\langle E \rangle) & = value(\langle T \rangle)\} \\ \langle T_0 \rangle ::= \langle T_1 \rangle \text{ `}*\text{' `}\mathbf{n}\text{'} & \{value(\langle T_0 \rangle) & = value(\langle T_1 \rangle) * eval(\text{`}\mathbf{n}\text{'})\} \\ \langle T \rangle ::= \text{`}\mathbf{n}\text{'} & \{value(\langle T \rangle) & = eval(\text{`}\mathbf{n}\text{'})\} \end{array}$$

The $+$ and $*$ operators appearing within braces correspond, respectively, to the standard operations of arithmetic addition and multiplication, and $eval(\cdot)$ evaluates its input as a number. Notice also that, within a production, different occurrences of the same grammar symbol are denoted by distinct subscripts.

Part II

Specification

Chapter 3

Analysis of Property Specification Patterns in SBAs

3.1 Overview

The concept of *pattern* has been initially proposed in the domain of architecture by Christopher Alexander [2], to represent:

> "*the description of a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*".

This idea of pattern has then been adopted in software engineering with the concept of *design patterns* [67], as reusable solutions for recurring problems in software design. Subsequently, the concept of design patterns has been embraced in different sub-domains of software engineering, from architectural patterns to reengineering patterns, including property specification patterns.

Property specification patterns [56] have been proposed in the late '90s in the context of finite-state verification, as a means to express recurring properties in a generalized form, which could be formalized in different specification languages, such as temporal logic. Specification patterns aimed at bridging the gap between finite-state verification tools (e.g., model checkers) and practitioners, by providing the latter with a powerful instrument for writing down properties to be fed to a formal verification tool. Given the origin of property specification patterns, most of past work has focused on the application of patterns to the specification (and the verification) of concurrent and real-time systems (for example, see [49]), with limited applications outside the research setting.

One of the questions that we asked ourselves during our research was whether existing requirements specification languages were expressive enough to formalize common requirements specifications used by SBAs practitioners. This led to the definition of a new research goal: evaluating the use of specification patterns for expressing properties of industrial SBAs, to assess whether existing and well-known specification patterns are adequate or not. If this is not the case, our next goal will become gathering substantial evidence for new specification patterns and/or language constructs required to support their practical use in industrial settings.

For these reasons, we conducted a study on the use of specification patterns in SBAs. The study has been performed by analyzing the requirements specifications of two sets of case studies. One set was composed of case studies extracted from research papers in the area of specification, verification and validation of SBAs, which appeared in the main publishing venues of software engineering and service-oriented computing within the last 10 years. The other set was composed of case studies corresponding to service interfaces written over a similar time period and used within the SBAs developed by our industrial partner, which operates in the banking domain.

During the analysis, we matched each SBA requirements specification against the patterns belonging to the specification pattern systems we selected from the research literature. When a match was not possible, we tried to classify the requirements specification according to a new pattern system, specific to the service provisioning domain, that we had been building during the matching process. Finally, we compared the results, in terms of matched patterns, for the research and the industrial case studies.

The chapter is organized as follows. Section 3.2 illustrates the specification patterns considered for the study. Section 3.3 describes the methodology used to conduct the study and presents its results, which are then discussed in section 3.4.

3.2 A Bird's Eye View of Specification Patterns

In this section we summarize the patterns we have used to classify the surveyed service specifications. We have categorized them in four groups: the first three groups correspond to systems of specification patterns well-known in the software engineering research community, but not necessarily used in the context of SBAs, while the last one includes patterns that are more specific to service provisioning. For each pattern we include a brief description as well as a simple property expressed using the pattern; in the sample properties, we use the letters P, S, T, and Z to denote events or states of a system execution.

The "D" Group

The first group corresponds to the property specification pattern system originally proposed by Dwyer et al. in [56]. This system includes nine parameterizable, high-level, formal specification abstractions. These patterns can be combined with five scopes ("global", "before", "after", "between", and "after until"), to indicate the portions of a system execution in which a certain pattern should hold. Note that in the rest of the chapter, we do not distinguish among the different scopes with which a certain pattern has been used, and report usage data aggregated over all possible scopes. The patterns are[1]:

Absence (D1) describes a portion of a system's execution that is free of certain events or states, as in "it is never the case that P holds".

Universality (D2) describes a portion of a system's execution that contains only states that have a desired property, as in "it is always the case that P holds".

Existence (D3) describes a portion of a system's execution that contains an instance of certain events or states, as in "P eventually holds".

Bounded existence (D4) describes a portion of a system's execution that contains at most a specified number of instances of a designated state transition or event, as in "it is always the case that the transitions to state P occur at most 2 times".

Precedence (D5) describes relationships between a pair of events or states, where the occurrence of the first is a necessary pre-condition for an occurrence of the second, as in "it is always the case that if P holds, then S previously held".

Response (D6) describes cause-effect relationships between a pair of events or states, where an occurrence of the first must be followed by an occurrence of the second, as in "it is always the case that if P holds, then S eventually holds".

Response chains (D7) is a generalization of the response pattern, as it describes relationships between *sequences* of individual states or events, as in "it is always the case that if P holds, and is succeeded by S, then T eventually holds after S".

Precedence chains (D8) is a generalization of the precedence pattern, as it describes relationships between *sequences* of individual states or events, as in "it is always the case that if P holds, then S previously held and was preceded by T".

Constrained chain patterns (D9) describes a variant of response and precedence chain patterns that restricts user specified events from occurring between pairs of states or events in the chain sequences, as in "it is always the case that if P holds, then S eventually holds and is succeeded by T where Z does not hold between S and T".

The "R" Group

The second group of patterns has been proposed by Konrad and Cheng [90] in the context of real-time specifications. This pattern system includes five patterns (and the same five scopes as in [56]) as well as a structured English grammar that supports both qualitative and real-time specification patterns. The five patterns are:

Minimum duration (R1) indicates the minimum amount of time a state formula has to hold once it becomes true, as in "it is always the case that once P becomes satisfied, it holds for at least k time units".

Maximum duration (R2) describes that a state formula always holds for less than a specified amount of time, as in "it is always the case that once P becomes satisfied, it

[1]A detailed description is available at `http://patterns.projects.cis.ksu.edu`.

holds for less than *k* time units".

Bounded recurrence (R3) indicates the amount of time in which a state formula has to hold at least once, as in "it is always the case that P holds at least every *k* time units".

Bounded response (R4) indicates the maximum amount of time that passes after a state formula holds until another state formula becomes true, as in "it is always the case that if P holds, then S holds after at most *k* time units".

Bounded invariance (R5) indicates the minimum amount of time a state formula must hold once another state formula is satisfied, as in "it is always the case that if P holds, then S holds for at least *k* time units".

The "G" Group

Another system of real-time specification patterns was developed, around the same time as the previous one, by Gruhn and Laue [73]. The system includes the actual patterns, certain types of combined events that can be used within specifications, and scopes that determine patterns validity. As for scopes, the authors support the possibility to express that a property holds before, after, and until a certain number of time units (possibly zero) have passed since the last occurrence of a certain event. The patterns are:

Time-bounded existence (G1) is the timed version of pattern D3, meaning that it can express properties such as "starting from the current point of time, P must occur within *k* time units".

Time-bounded response (G2) represents the same pattern as R4.

Precedence with delay (G3) represents, together with the next pattern, the timed version of pattern D5. In this first variant, it can state properties such as "P must always be preceded by S and at least *k* time units have passed since the occurrence of S".

Time-restricted precedence (G4) is the second timed variant of pattern D5; it can express properties such as "P must always be preceded by S and must occur within at most *k* time units since the occurrence of S".

The "S" Group

This group combines the patterns we found in the literature dealing with SBAs specifications, which do not appear in the pattern systems described above; for this reason, we group them all together under the *service provisioning* patterns label.

Average response time (S1) is a variant of the bounded response pattern (R4) that uses the average operator to aggregate the response time over a certain time window.

Counting the number of events (S2) is used (see, for example, [129]) to express common non-functional requirements such as reliability (e.g., "number of errors in a given time window") and throughput (e.g., "number of requests that a client is allowed to submit in a given time window").

Average number of events (S3) is a variant of the previous pattern that states the average number of events occurred in a certain time interval within a certain time window, as in "the average number of client requests per hour computed over the daily business hours".

Maximum number of events (S4) is another variant of the S2 pattern that aggregates events using the maximum operator, as in "the maximum number of client requests per hour computed over the daily business hours".

Absolute time (S5) indicates events that should occur at a time that satisfies an absolute time constraint, as in "if the booking is done in the first week of March, a discount is given" (taken from [85]).

Unbounded Elapsed time (S6) indicates the time elapsed since the last occurrence of a certain event.

Data-awareness (S7) is a pattern denoting properties that refer to the actual data content of messages exchanged between services as in "every ID present in a message cannot appear in any future message" (taken from [76]).

Note that patterns S1–S4 express aggregate statistics, without assuming any underlying probabilistic model. Moreover, pattern S7 is usually used orthogonally in combination with other patterns, to create their data-aware versions.

3.3 The Survey

In our study, we extracted specification patterns for SBAs by analyzing examples and case studies both from the research literature and from industry.

We analyzed the requirements specifications for the SBA(s) described in each example or case study, and manually classified each specification to match the patterns defined in the previous section. The specifications were in many forms: some were expressed using a specification formalism (e.g., a temporal logic), while others were expressed in English. When a specification could not be easily matched with a pattern, we used the criteria proposed in [57] to still count a specification as a match: a) formal equivalence; b) equivalence by parameter substitution; c) variant of a pattern; d) wrong formal specification with matching prose description.

Note that a single requirements specification may match more than one pattern; for example, a property like:

> *if a message with a red code alert is received three times for the same patient during a time span of a week, then doctors should send a confirmation for the hospitalization of that patient within an hour from the reception of the last alert message*" (adapted from [9])

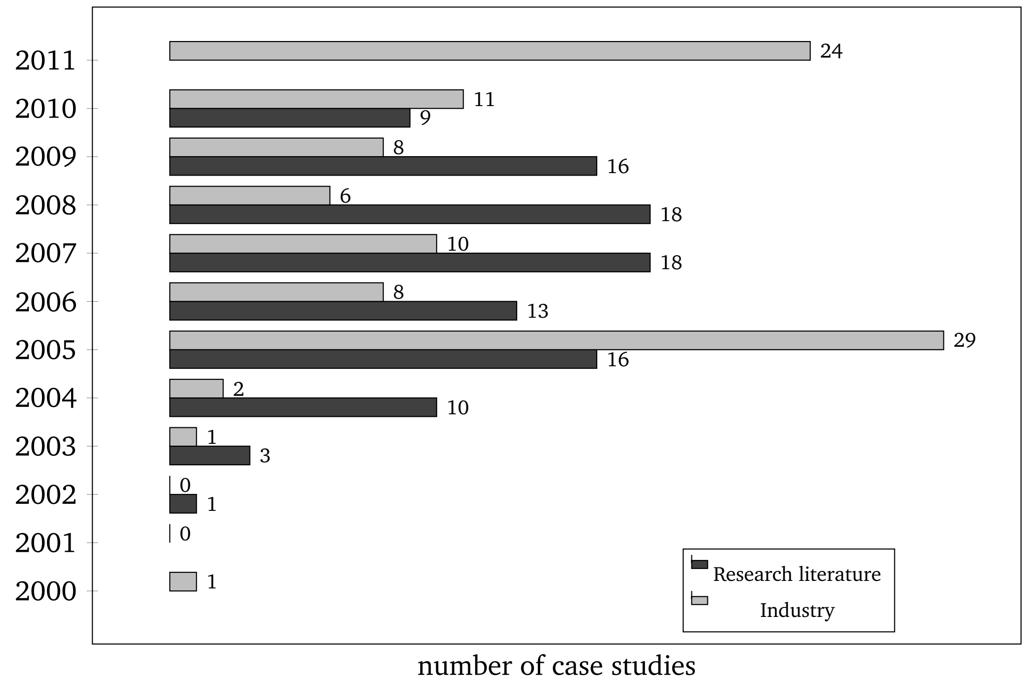


Figure 3.1. **Number of case studies considered per year**

is an instantiation of patterns R4 (*bounded response time*), S2 (*counting*) and S7 (*data-awareness*).

The set of case studies we considered spans over more than ten years, as shown in figure 3.1. Overall, we considered 104 case studies from the research literature and 100 industrial ones. In the rest of this section we describe, for each of the two categories of case studies, the data sources and the data themselves.

3.3.1 Research Literature Data

The research case studies have been extracted from papers published between 2002 and 2010; the reason for choosing 2002 as the left bound is that research in the area of (Web) SBAs originated around that time. As publication venues to analyze, we considered the main conferences in software engineering (ASE, FASE, SIGSOFT FSE, ICSE), the main conferences in service-oriented computing (ECOWS, ICSOC, ICWS, SCC, SERVICES, SOCA, WS-FM, WWW), the major journals in the two areas (respectively, ACM TOSEM and IEEE TSE for software engineering, and ACM TWEB and IEEE TSC for service-oriented computing). For each of these venues, we selected papers on specification, validation, and verification of SBAs; from this set, subsequently, we only considered papers with at least one case study with at least one requirements speci-

Table 3.1. **Number of papers considered, per scientific venue, per year**

venue	2002	2003	2004	2005	2006	2007	2008	2009	2010
ASE	0	1	1	0	0	0	0	0	0
ECOWS	–	0	0	1	0	2	0	2	1
FASE	0	0	0	1	0	0	1	0	0
FSE	0	0	0	0	0	0	1	1	1
ICSE	0	0	1	0	0	0	1	0	0
ICSOC	–	0	3	3	0	2	1	1	1
ICWS	–	0	1	1	4	3	2	2	1
SCC	–	–	0	1	0	1	1	1	0
SERVICES	–	–	–	–	–	–	0	0	0
SOCA	–	–	–	–	–	1	–	0	0
WSFM	–	–	1	1	2	2	2	1	0
WWW	0	0	1	1	1	0	0	0	0
TOSEM	0	0	0	0	0	0	0	0	1
TSE	0	0	0	0	0	0	0	2	1
TWEB	–	–	–	–	–	0	0	0	1
TSC	–	–	–	–	–	–	0	2	0
other	1	2	2	5	8	7	5	3	2
total	1	3	10	14	15	18	14	15	9

fication[2]. Moreover, we also included other papers on specification, verification, and verification of SBAs that we were aware of and that had appeared in other venues; however, these venues have not been systematically surveyed. An overview of the number of papers considered, for each venue, is shown in table 3.1; note that the values displayed in the table on the row labeled “total” do not match the values shown in figure 3.1 because in some cases the same paper illustrated more than one case study.

Although we analyzed 104 case studies, we counted only 36 distinct examples, i.e., in many cases, the same example has been used in different case studies across various papers. The top four recurring examples are “loan approval” (13 times), “travel agency” (12 times), “online shopping” (11 times), and “car rental” (8 times).

Out of these case studies, we analyzed and classified 290 requirements specifications. We successfully matched 272 specifications against the patterns presented in section 3.2; the pattern usage distribution is shown in table 3.2.

A portion of these data (the group corresponding to patterns D1–D8, representing the 63% of the specifications) can be compared with existing data available in literature. Indeed, reference [57] presents the usage frequency for patterns in the “D”

[2]In few cases, we also considered papers that included at least one requirements specification formulated in a general way, i.e., not related to a specific example or case study.

Table 3.2. **Patterns usage in research specifications**

pattern	occurrence	distribution %
D6	76	27.9
R4	52	19.1
S7	47	17.3
D5	22	8.1
D1	20	7.4
S2	20	7.4
D2	19	7
D3	17	6.3
D7	8	2.9
D8	6	2.2
S1	5	1.8
D4	3	1.1
G1	2	0.7
S3	2	0.7
S5	2	0.7
S6	2	0.7
R3	1	0.4
G3	1	0.4
D9	0	0
R1	0	0
R2	0	0
R5	0	0
G4	0	0
S4	0	0

group, extracted from a set of 511 matched specifications belonging to various application domains, such as hardware and communication protocols, control systems, and distributed object systems. The comparison of our data for patterns D1–D8, with respect to the data presented in [57] is shown in figure 3.2. Despite different rankings, the five most common patterns are the same (D1, D2, D3, D5, and D6); moreover, the most common pattern is D6 (*response*), with a similar usage frequency in both data sets.

3.3.2 Industrial Data

The industrial case studies have been provided by our industrial partner Credit Suisse, a world-leading financial services company headquartered in Switzerland.

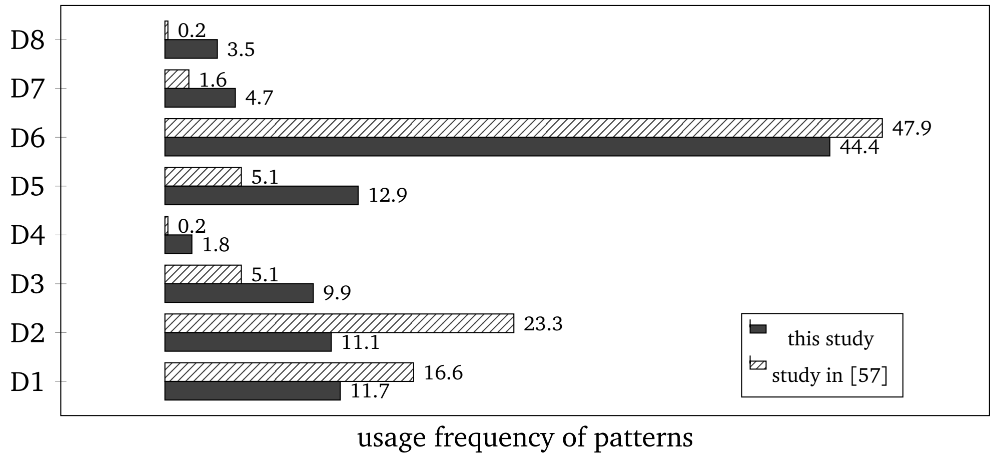


Figure 3.2. **Comparison of the usage frequency of patterns of the "D" group in research case studies, as reported by our study and by reference [57]**

Credit Suisse started to implement an SOA in 2000, as a means to leverage its encompassing set of "legacy" mainframe IT applications. In the process, Credit Suisse has established one of the largest CORBA-based service backbones in industry, which has recently been extended to support Web services standards [92]. Credit Suisse operates the Interface Management System (IFMS) as a central information base for all service interfaces available for reuse [116]. IFMS is an integral part of an application developer's work process: not only it does provide documentation on interfaces, but it also generates the required code artifacts (service stubs) to use a service. For new service interfaces, IFMS provides a workflow covering all tasks related to definition, specification, and quality management, thus linking the staff involved during the phases of service development, testing and deployment.

The service specifications analyzed in this study were extracted from IFMS by selecting a random subset of 100 service interfaces. They cover the whole range of application domains at Credit Suisse, such as accounts, payments, customers, financial securities operations, and stock exchange. When an interface contained multiple versions of a service, we extracted specifications from the most recent version. The selected interfaces have been defined between 2000 and 2011.

The general structure of an interface document includes, among others, sections about pre- and post-conditions of the service, as well as on non-functional assertions under different usage conditions; we extracted requirement specifications from all these sections, when available.

In total, we extracted 625 requirements specifications from the set of 100 case studies. We matched 562 of them against the patterns presented in section 3.2; the pattern usage distribution is shown in table 3.3.

Table 3.3. **Patterns usage in industrial specifications**

pattern	occurrence	distribution %
S3	201	35.8
S4	168	29.9
S7	97	17.3
S1	91	16.2
D6	11	2
D1	1	0.2
D2	0	0
D3	0	0
D4	0	0
D5	0	0
D7	0	0
D8	0	0
D9	0	0
R1	0	0
R2	0	0
R3	0	0
R4	0	0
R5	0	0
G1	0	0
G3	0	0
G4	0	0
S2	0	0
S5	0	0
S6	0	0

3.4 Discussion

To compare them, the pattern usage distributions of tables 3.2 and 3.3 have been combined and plotted on the chart displayed in figure 3.3. It is possible to immediately see the discrepancy of pattern usage between research and industrial case studies, with a separation line virtually drawn before the patterns of the “S” group.

It is clear that the majority of requirements specifications used in industrial settings matches the S1, S3, S4 and S7 patterns; below, we discuss the usage of each of these patterns in the two categories.

As for pattern S1 (“average response time”), we have already stated that it can be considered a variant of pattern R4 (“response time”); moreover, R4 is the second most used pattern in the specifications from the research literature. In light of this, we can

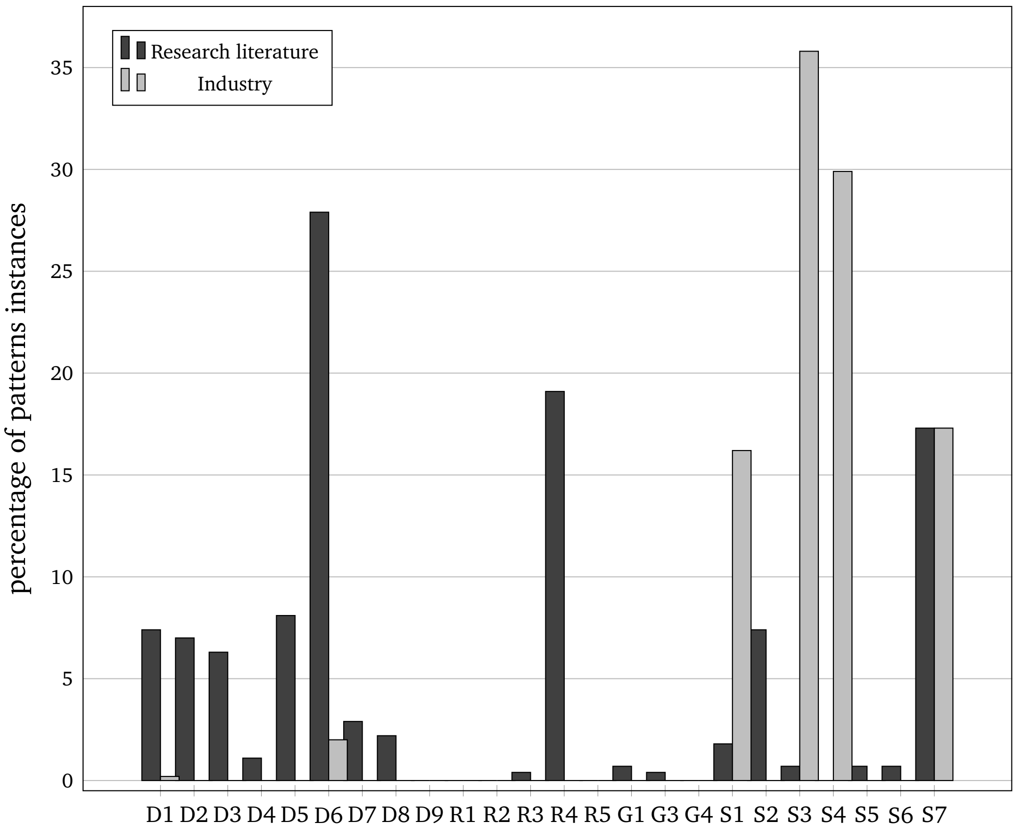


Figure 3.3. **Comparison of patterns usage (percentage) between the research and the industrial case studies**

compare the usage of pattern S1 (16.2%) in industrial specifications with the combined usage of patterns S1 and R4 (20.9%) in the ones from research literature; furthermore, we should also consider that pattern R4 is not used at all in industrial specifications. It is evident that the concept of response time has the same importance, in terms of relevance for the specifications, in both categories of specifications. However, while this concept is used exclusively in its aggregated form (through the average operator) in the industrial specifications, this is not true for research case studies, where the aggregate variant has been used only in five properties (found across five papers).

Similar observations can be made by comparing the usage of patterns S3 and S4 (respectively, “average” and “maximum number of events”), since they represent aggregated variants of pattern S2 (“counting the number of events”). As for the other pattern considered above, it is evident that industrial specifications use only aggregated variants (through the average and maximum operators) of the concept represented by pattern S2. Moreover, aggregated variants of pattern S2 are used very rarely in research case studies; in this case, only pattern S3 is used, and only in two properties (across two papers, from the same authors). Another point to consider is that while counting patterns such as S3 and S4 represent the majority (65.7%) of industrial specifications, the combined usage of patterns S2 and S3 in research specifications is only 8.1%.

As for pattern S7 (“data-awareness”), the figure (17.3%) of its usage in both set of case studies is the same. Indeed, we have noticed in both sets of specifications that this pattern is often used to state pre-/post-conditions on data exchanged by a service. We also note that some recent research (for example, see [76]) has investigated support for data-aware properties in specification languages such as temporal logics, representing a good example of an industrial need met by academic research.

Finally, the remaining patterns matched by industrial specifications have been D1 (“absence”), matched only once, and D6 (“response”), matched eleven times. These two patterns actually represent the only patterns matched from the “D”, “R” and “G” groups within the set of industrial case studies.

All the observations made above imply two main points:

- The majority of requirements specifications stated in industrial settings refers to non-functional properties expressed using aggregate operators (e.g., average, count, maximum). Similar requirements are found only rarely in the research literature and when so, they are expressed using the non-aggregated versions of the patterns.

- The specification patterns proposed in the research literature are barely used in industrial settings. This may be an indication either of the lack of need for expressing such properties within industrial specifications or of the need for technology transfer in the area of requirements specification languages.

3.5 Summary

The study illustrated in this chapter compared the usage of property specification patterns in requirements specifications of SBAs, between research and industrial case studies. The study showed that: a) the majority of requirements specifications stated in industrial settings refers to specific aspects of service provisioning, which can be characterized as a new class of specification patterns; b) the specification patterns proposed in the research literature are barely used in industrial settings.

These considerations indicate that some needs of the industry are not fully met by software engineering research. This suggests that new research directions in the areas of requirements specification languages and of the related verification techniques should be explored. In the following chapter, we present our response to the first need, in terms of a new specification language for SBAs.

Chapter 4

The SOLOIST Specification Language

4.1 Overview

This chapter introduces SOLOIST (*SpecificatiOn Language fOr servIce compoSitions inTeractions*), our language to specify properties of service compositions. SOLOIST builds upon our previous experience on defining specification languages for service compositions; see, for example, Timed WS-CoL [8] and ALBERT [9]. In a certain sense, SOLOIST can be seen as a profound revision of our previous attempts, designed on the basis of the results of the study presented in the previous chapter. Our main goal has been to design a formal language that is both expressive—to meet the requirements raised from our field study—and, at the same time, usable with techniques and tools for automated verification.

The chapter is structured as follows. Section 4.2 discusses some design choices made during the definition of the language. Section 4.3 introduces SOLOIST, its syntax, and its semantics (both informally and formally). Section 4.4 shows the use of the language to specify some properties of a BPEL process. Section 4.5 illustrates the translation of SOLOIST into linear temporal logic.

4.2 Language Design

Our starting point has been a temporal logic with metrics: this allows us to support the patterns of the "D" [56], "R" [90], and "G" [73] groups, i.e., the ones prescribing constraints on the order and/or the occurrence of events, possibly with (real-)time information. The logic assumes a discrete time domain, with each occurrence of an event denoted by a timestamp.

As for supporting the *service provisioning* patterns ("S" group), we made different decisions. First, we decided not to support patterns referring to absolute or elapsed time (patterns S5 and S6), since this would have notably impacted on the complexity of the translation. Moreover, our field study showed that both of them are used in less than 1% of the specifications; given these data, we maintain this decision does not critically affect the expressiveness of the language as well as its reception by practitioners.

Pattern S7 is supported by adding a first-order quantification to the logic, following the approach proposed in [76]. By making the simplifying assumption that domains over which the quantification ranges are finite, the first-order quantification is mere syntactic sugar, which does not impact on the decidability of the language, but helps to improve its readability. The logic is also many-sorted, to support the different types of the messages exchanged among services.

Regarding patterns S3 and S4, which define properties related to the aggregation of events occurred in a certain time interval h within a certain time window K as in "the average (maximum) number of service invocations per hour over the last 11.5 hours of operation", we run into different possibilities to represent the observation interval h (i.e., one hour in the example) within the time window K (i.e., 11.5 hours in the example) considered to compute the aggregate value. It could be defined either as a fixed window over adjacent, non-overlapping intervals, or as a sliding window over overlapping intervals. The latter interpretation would require also to define a minimal distance corresponding to the shift of the sliding window, which could be either a fixed value, such as a system tick, or a variable value, such as the timestamp of each event occurrence (meaning that the window slides variably, according to the occurrences of the events). Furthermore, in both interpretations, one has to make a decision on how to deal with time windows whose length is not an exact multiple of the observation interval; in other words, how to consider the tail of the window whose length is less than the one of the observation interval. This is the case for the property mentioned above: assuming that a time unit corresponds to a minute, we have an observation interval long 60 time units and a time window long 690 time units; the tail of the window is then $690 \bmod 60 = 30$. After consulting with our industrial partner and evaluating its needs, we decided to support the interpretation with adjacent, non-overlapping observation intervals, where tail intervals whose length is shorter then the observation interval are ignored to express pattern S3 but considered to express pattern S4.

Modeling pattern S2 was straightforward, while for pattern S1 we considered its specific use in the context of SBAs. It shall be used to specify the average response time of invocations made to a certain service over a certain time window. Since a service may provide multiple operations, we decided to include the possibility to specify which operations to consider when computing the aggregate response time, as well as the calling points within the workflow of a service composition from which the invocations originate. Moreover, every service invocation in the scope of an instance of pattern S1 is assumed to be synchronous and actually corresponding to a pair of events, the *start* and *end* one. These events corresponds to the start (end) of an invocation in a precise location of the workflow; a start (end) of an invocation to the same operation of a service but from a different location in the workflow is considered a distinct event. Under these premises, we assume that two subsequent occurrences of the same *start* or *end* event may not happen.

4.3 The Language

4.3.1 Preliminaries

A signature Σ is a tuple $\langle S; F; P\rangle$ where:

- S is a set of sort symbols, i.e., names representing various domains;
- F is a set of pairs $\langle f; s_1 \times \ldots \times s_n \rightarrow w\rangle$ where $n \geq 0$, f is a function symbol, $s_1 \times \ldots \times s_n \rightarrow w$ is the type of f, and $s_1, \ldots, s_n, w \in S$;
- P is a set of pairs $\langle p; s_1 \times \ldots \times s_n\rangle$ where $n \geq 0$, p is predicate symbol, $s_1 \times \ldots \times s_n$ is the type of p, and $s_1, \ldots, s_n \in S$.

The sets S, F, P of Σ are denoted by $Sort(\Sigma), Func(\Sigma), Pred(\Sigma)$. Notice that constants are modeled as nullary functions of the form $c :\rightarrow w$.

Let Σ be a signature. For each sort $s \in Sort(\Sigma)$, we assume a set V_s of variables of sort s disjoint from the constants in $Func(\Sigma)$. Also, for each sort $s \in S$, we define the set of terms of sort s by induction:

- a variable $x \in V_s$ of sort s is a term of type s;
- if $f : s_1 \times \ldots \times s_n \rightarrow w \in Func(\Sigma)$ and $t_1, \ldots, t_n$ are terms of type $s_1, \ldots, s_n$ respectively, than $f(t_1, \ldots, t_n)$ is a term of type w.

An atom has the form $p(t_1, \ldots, t_n)$, with $p(s_1, \ldots, s_n) \in Pred(\Sigma)$ and terms $t_1, \ldots, t_n$ of type $s_1, \ldots, s_n$.

4.3.2 Syntax

A SOLOIST formula over Σ is defined inductively as follows:

- if $t_1, \ldots, t_n$ are terms of type $s_1, \ldots, s_n$ and $p(s_1, \ldots, s_n) \in Pred(\Sigma)$ is a predicate symbol, then $p(t_1, \ldots, t_n)$ is a formula;
- if ϕ and ψ are formulae and x is a variable, then $\neg\phi$, $\phi \wedge \psi$, $\exists x : \phi$ are formulae;
- if ϕ and ψ are formulae and I is a nonempty interval over $\mathbb{N}$, then $\phi \mathsf{U}_I \psi$ and $\phi \mathsf{S}_I \psi$ are formulae;

- if $n, K \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >, =\}$, ϕ is a formula of the form $p(t_1, \dots, t_n)$, with $p(s_1, \dots, s_n) \in \mathit{Pred}(\Sigma)$ and terms $t_1, \dots, t_n$ of type $s_1, \dots, s_n$, then $\mathsf{C}^K_{\bowtie n}(\phi)$ is a formula;
- if $n, K, h \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >, =\}$, ϕ is a formula of the form $p(t_1, \dots, t_n)$, with $p(s_1, \dots, s_n) \in \mathit{Pred}(\Sigma)$ and terms $t_1, \dots, t_n$ of type $s_1, \dots, s_n$, then $\mathsf{V}^{K,h}_{\bowtie n}(\phi)$ and $\mathsf{M}^{K,h}_{\bowtie n}(\phi)$ are formulae;
- if $n, K \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >, =\}$, $\phi_1, \dots, \phi_m, \psi_1, \dots, \psi_m$ are formulae of the form $p(t_1, \dots, t_n)$—with $p(s_1, \dots, s_n) \in \mathit{Pred}(\Sigma)$ and terms $t_1, \dots, t_n$ of type $s_1, \dots, s_n$—where for all $i, 1 \leq i \leq n, \phi_i \neq \psi_i$, then $\mathsf{D}^K_{\bowtie n}\{(\phi_1, \psi_1), \dots, (\phi_m, \psi_m)\}$ is a formula.

Additional temporal modalities can be defined from the U_I and S_I modalities using the usual conventions. Note that the arguments of modalities $\mathsf{C}, \mathsf{V}, \mathsf{M}, \mathsf{D}$ can only be atoms, i.e., positive literals; this reflects the fact that they represent the occurrences of certain events, which are then aggregated as prescribed by the modality.

4.3.3 Informal Semantics

The informal semantics of SOLOIST is based on a sequence of timestamped predicates. A predicate corresponds to an event, which models the execution of an activity defined within a service composition; its arguments are the parameters possibly associated with the activity, such as the input message of a service invocation.

The U_I and S_I modalities have the usual meaning in temporal logics ("*Until*" and "*Since*")[1].

The $\mathsf{C}^K_{\bowtie n}(\phi)$ modality, evaluated in a certain time instant, states a bound on the number of occurrences of an event ϕ, counted over a time window K; it expresses pattern S2.

The $\mathsf{V}^{K,h}_{\bowtie n}(\phi)$ modality, evaluated at a certain time instant τ_i, is used to express a bound on the average number (with respect to an observation interval h, open to left and closed to the right) of occurrences of an event ϕ, occurred within a time window K; this corresponds to pattern S3. As discussed in section 4.2, since K may not be an exact multiple of h, the actual time window over which occurrences of event ϕ are counted is bounded by $\tau_i - \lfloor \frac{K}{h} \rfloor h$ on the left and by τ_i on the right; similarly, the number of observation intervals taken into account to compute the average is $\lfloor \frac{K}{h} \rfloor$. Consider, for example, the sequence of events depicted in figure 4.1, where black circles correspond to occurrences of the ϕ event. Assuming $\tau_i = 42$, $K = 35$, and $h = 6$ (values expressed as time units), $\lfloor \frac{K}{h} \rfloor = \lfloor \frac{35}{6} \rfloor = 5$. The evaluation of the formula $\mathsf{V}^{35,6}_{\bowtie n}(\phi)$ at time instant 42 is then $\frac{2+1+2+4+1}{5} \bowtie n$, where the numerator of the fraction to the left of $\bowtie$ is the number of event occurrences in the window bounded by τ_i and $\tau_i - 5h$.

[1]As will be shown in the next subsection, a strict semantics is assumed for the U_I and S_I modalities.

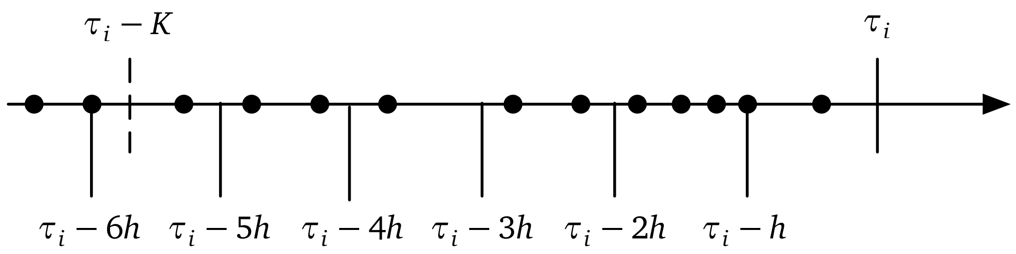


Figure 4.1. **Sequence of events over a time window K, with observation interval h (semantics of the V and M modalities)**

The $\mathsf{M}^{K,h}_{\bowtie n}(\phi)$ modality, evaluated in a certain time instant τ_i, is used to express a bound on the maximum number (with respect to an observation interval h, open to left and closed to the right) of occurrences of an event ϕ, occurred within a time window K; this corresponds to pattern S4. Differently from the V modality described above, this modality takes also into account the events occurring in a tail interval, even if its length is shorter than the one of the observation interval h. With reference to figure 4.1 and assuming the same values as above for τ_i, K, and h, the tail interval bounded by $\tau_i - K$ on the left and $\tau_i - \lfloor \frac{K}{h} \rfloor h = \tau_i - 5h$ on the right is also considered for computing the aggregate value. This leads to a final evaluation for the formula equivalent to $\max(\{1\} \cup \{4\} \cup \{2\} \cup \{1\} \cup \{2\} \cup \{1\}) \bowtie n = 4 \bowtie n$, where the i-th singleton set in the argument of the aggregate operator corresponds to the number of event occurrences in the i-th observation interval within the time window.

The D modality, evaluated in a certain time window τ_i, expresses a bound on the average time elapsed between pairs of specific adjacent events, occurred within a time window K; it can be used to express pattern S1. Consider, for example, the sequence of events depicted in figure 4.2, where capital letters in the lower part of the timeline correspond to events, and numbers in the upper part of the timeline indicate time-

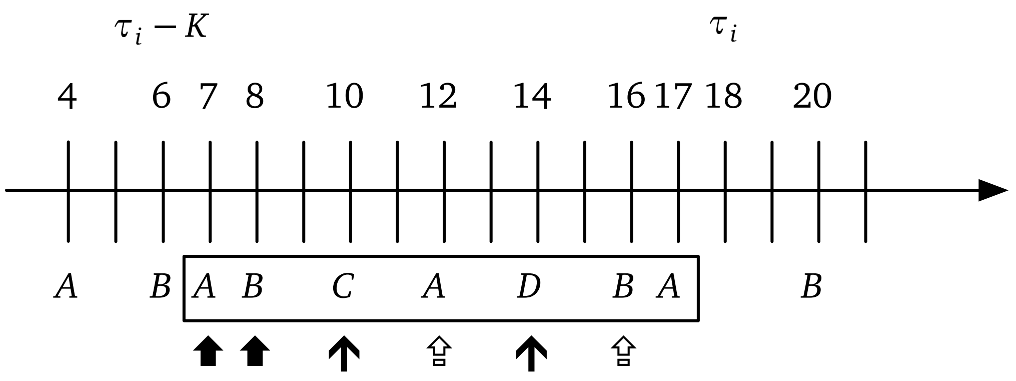


Figure 4.2. **Sequence of pairs of events over a time window K (semantics of the D modality)**

stamps; assume that the current time instant is $\tau_i = 18$ and that $K = 12$. To express a bound for the average distance between each occurrence of an event A and the first subsequent occurrence of an event B, as well as for the pair of events (C, D), for the previous 12 time units, one writes a formula like $\mathsf{D}^{12}_{\bowtie n}\{(A,B),(C,D)\}$, for some $\bowtie$ and n. With respect to $\tau_i = 18$, the time window of length $K = 12$ includes the events (with their respective timestamp) $(A,7)$, $(B,\ 8)$, $(C,\ 10)$, $(A,\ 12)$, $(D,\ 14)$, $(B,\ 16)$, $(A,17)$, enclosed in the rectangle in figure 4.2. The average time distance is then computed by summing the differences between the timestamps of each (A,B) and (C,D) pair (each pair of events is denoted by a different kind of arrow in figure 4.2), and dividing the result for the number of the selected events pairs (3 in the example). Finally, the D modality compares this result with value n, according to the relation defined by $\bowtie$; i.e., the evaluation of $\mathsf{D}^{12}_{\bowtie n}\{(A,B),(C,D)\}$ is $\frac{(8-7)+(16-12)+(14-10)}{3} \bowtie n$. Note that the event $(A,17)$ is ignored for computing the (average) distance, since it is not matched by a corresponding B event within the selected time window.

4.3.4 Formal Semantics

A Σ-structure associates appropriate values to the elements of a signature Σ. A Σ-structure $\mathscr{D}$ consists of:

- a non-empty set $s^{\mathscr{D}}$ for each sort $s \in \mathit{Sort}(\Sigma)$;
- a function $f^{\mathscr{D}} : s_1^{\mathscr{D}} \times \ldots \times s_n^{\mathscr{D}} \to w^{\mathscr{D}}$ for each function symbol $f : s_1 \times \ldots \times s_n \to w \in \mathit{Func}(\Sigma)$;
- a relation $p^{\mathscr{D}} \subseteq s_1^{\mathscr{D}} \times \ldots \times s_n^{\mathscr{D}}$ for each predicate symbol $p : s_1 \times \ldots \times s_n \in \mathit{Pred}(\Sigma)$;

A *temporal first-order* structure over Σ is a pair $(\bar{\mathscr{D}}, \bar{\tau})$, where $\bar{\mathscr{D}} = \mathscr{D}_0, \mathscr{D}_1, \ldots$ is a sequence of Σ-structures and $\bar{\tau} = \tau_0, \tau_1, \ldots$ is a sequence of natural numbers (i.e., timestamps), where:

- the sequence $\bar{\tau}$ is monotonically increasing (i.e., $\tau_i < \tau_{i+1}$, for all $i \geq 0$);
- for each $\mathscr{D}_i$ in $\bar{\mathscr{D}}$, with $i \geq 0$, for each $s \in \mathit{Sort}(\Sigma)$, $s^{\mathscr{D}_i} = s^{\mathscr{D}_{i+1}}$;
- for each $\mathscr{D}_i$ in $\bar{\mathscr{D}}$, with $i \geq 0$, for each function symbol $f \in \mathit{Func}(\Sigma)$, $f^{\mathscr{D}_i} = f^{\mathscr{D}_{i+1}}$.

A variable assignment σ is a $\mathit{Sort}(\Sigma)$-indexed family of functions $\sigma_s : V_s \to s^{\mathscr{D}}$ that maps every variable $x \in V_s$ of sort s to an element $\sigma_s(x) \in s^{\mathscr{D}}$. Notation $\sigma[x/d]$ denotes the variable assignment that maps x to d and maps all other variables as σ does.

The valuation function $[\![t]\!]^{\mathscr{D}}_{\sigma}$ of term t for a Σ-structure $\mathscr{D}$ is defined inductively as follows:

- if t is a variable $x \in V_s$, then $[\![t]\!]^{\mathcal{D}}_{\sigma} = \sigma_s(x)$;
- if t is a term $f(t_1, \dots, t_n)$ then $[\![t]\!]^{\mathcal{D}}_{\sigma} = f^{\mathcal{D}}([\![t_1]\!]^{\mathcal{D}}_{\sigma}, \dots, [\![t_n]\!]^{\mathcal{D}}_{\sigma})$.

For the sake of readability, we drop the superscript $\mathcal{D}$ and the subscript σ from the valuation function $[\![\cdot]\!]$ when they are clear from the context.

Given a temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$ over Σ, a variable assignment σ, symbols $i, n, K, h \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >, =\}$, we define the satisfiability relation $(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \phi$ for SOLOIST formulae as depicted in figure 4.3.

$$
\begin{array}{lll}
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models p(t_1, \dots, t_n) & \text{iff} & ([\![t_1]\!], \dots, [\![t_n]\!]) \in p^{\mathcal{D}_i} \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \neg\phi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \not\models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \phi \wedge \psi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \phi \wedge (\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \exists x \colon \phi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, \sigma[x/d], i) \models \phi \\
 & & \text{for some } d \in s^{\mathcal{D}} \text{ (with } x \text{ of sort } s) \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \phi \mathsf{S}_I \psi & \text{iff} & \text{for some } j < i, \tau_i - \tau_j \in I, \\
 & & (\bar{\mathcal{D}}, \bar{\tau}, \sigma, j) \models \psi \text{ and} \\
 & & \text{for all } k, j < k < i, (\bar{\mathcal{D}}, \bar{\tau}, \sigma, k) \models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \phi \mathsf{U}_I \psi & \text{iff} & \text{for some } j > i, \tau_j - \tau_i \in I, \\
 & & (\bar{\mathcal{D}}, \bar{\tau}, \sigma, j) \models \psi \text{ and} \\
 & & \text{for all } k, i < k < j, (\bar{\mathcal{D}}, \bar{\tau}, \sigma, k) \models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models C^{K}_{\bowtie n}(\phi) & \text{iff} & c(\tau_i - K, \tau_i, \phi) \bowtie n \text{ and } \tau_i \geq K \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \mathsf{V}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \dfrac{c(\tau_i - \lfloor \frac{K}{h} \rfloor h, \tau_i, \phi)}{\lfloor \frac{K}{h} \rfloor} \bowtie n \text{ and } \tau_i \geq K \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \mathsf{M}^{K,h}_{\bowtie n}(\phi) & \text{iff} & \max\{aggrc(m, K, h, \phi)\} \bowtie n \text{ with} \\
 & & lb(m) = \max\{\tau_i - K, \tau_i - (m+1)h\} \\
 & & \text{and } rb(m) = \tau_i - mh, \text{ with } \tau_i \geq K \\
(\bar{\mathcal{D}}, \bar{\tau}, \sigma, i) \models \mathsf{D}^{K}_{\bowtie n}\{(\phi_1, \psi_1), \dots, (\phi_m, \psi_m)\} & \text{iff} & \dfrac{\sum_{j=1}^{m} \sum_{(s,t) \in d(\phi_j, \psi_j, \tau_i, K)} (\tau_t - \tau_s)}{\sum_{j=1}^{m} |d(\phi_j, \psi_j, \tau_i, K)|} \bowtie n \\
 & & \text{with } \tau_i \geq K
\end{array}
$$

where
$c(\tau_a, \tau_b, \phi) = |\{s \mid \tau_a < \tau_s \leq \tau_b \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, \sigma, s) \models \phi\}|$,
$aggrc(m, K, h, \phi) = \bigcup_{m=0}^{\lfloor \frac{K}{h} \rfloor} \{c(lb(m), rb(m), \phi)\}$, and
$d(\phi, \psi, \tau_i, K) =$
$\{(s,t) \mid \tau_i - K < \tau_s \leq \tau_i \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, \sigma, s) \models \phi, t = \min\{u \mid \tau_s < \tau_u \leq \tau_i, (\bar{\mathcal{D}}, \bar{\tau}, \sigma, u) \models \psi\}\}$.

Figure 4.3. **Formal semantics of SOLOIST**

4.4 SOLOIST at Work

In this section we show how SOLOIST can be used to specify properties related to the interactions of a service composition described in BPEL.

Let $\mathscr{A}$ be the set of activities defined in a BPEL process[2]; $\mathscr{A} = \mathscr{A}_{start-inv} \cup \mathscr{A}_{end-inv} \cup \mathscr{A}_{recv} \cup \mathscr{A}_{pick} \cup \mathscr{A}_{reply} \cup \mathscr{A}_{hdlr} \cup \mathscr{A}_{other}$ where:

- $\mathscr{A}_{start-inv}$ ($\mathscr{A}_{end-inv}$) is the set of *start* (*end*) events of all *invoke* activities[3];
- $\mathscr{A}_{recv}$ is the set of all *receive* activities;
- $\mathscr{A}_{pick}$ is the set of all *pick* activities;
- $\mathscr{A}_{reply}$ is the set of all *reply* activities;
- $\mathscr{A}_{hdlr}$ is the set of events associated with all kinds of *handlers*;
- $\mathscr{A}_{other}$ is the set of activities that are not an *invoke*, a *receive*, a *pick*, a *reply*, or related to a handler (e.g., an *assign*, a control structure activity).

Let $\mathscr{A}_{msg} = \mathscr{A} \setminus \mathscr{A}_{other}$ be the set of activities that involve a data exchange, i.e., that have either an input message or an output message attached with them. Each $\mu \in \mathscr{A}_{msg}$ has an arity corresponding to the sum of the simple type variables by which its input and output messages can be represented; each $\mu \in \mathscr{A}_{other}$ is nullary.

A signature Σ to specify the interactions of a BPEL process with partner services by means of SOLOIST can be defined as follows:

- S is the set of XML simple types (e.g., integer, character, string);
- F is the set of functions defined by the scripting language used within the process (e.g., XPath functions on integers and strings);
- $P = \mathscr{A}$. A predicate may correspond to the execution of an activity; its arity and type are then those of the corresponding activity. The usage of the equality predicate between terms of the same XML type is also allowed.

Following the definitions in section 4.3, the variables of a BPEL process are partitioned into various domains V_s, with $s \in Sort(\Sigma)$.

We assume that the process has an integer variable `foo`, an *invoke* activity named *invA* that takes and returns an integer, an *invoke* activity named *invB* with no input or output parameters, three *receive* activities named *recvP*, *recvQ*, and *recvR* and a reply activity *term* that takes no parameters. The detailed workflow structure of the process as well as the other variables are of no interest for the purpose of this section and are omitted for clarity.

Below we list some properties associated with this process and expressed in natural language, followed by their translation into SOLOIST formulae. All properties are under the scope of an implicit universal temporal quantification as in *"In every process run, . . . "*.

[2]Activities of a BPEL process can be uniquely identified by means of an XPath expression.

[3]A synchronous *invoke* is characterized both by a *start* event and by an *end* event; an asynchronous *invoke* is characterized only by a *start* event.

1. *"At the end of the execution of the activity* invA*, the value of variable* `foo` *should be equal to 42."*
   $\mathsf{G}(\forall x, y : invA_{end}(x, y) \rightarrow \mathtt{foo} = 42)$

2. *"The execution of activity* recvP *should alternate with the execution of activity* recvQ*, though other activities different from* recvQ *(respectively,* recvP*) can be executed in between."*
   $\mathsf{G}((recvP \rightarrow \neg recvP \mathsf{U}_{(0,\infty)} recvQ) \wedge (recvQ \rightarrow \neg recvQ \mathsf{U}_{(0,\infty)} recvP))$

3. *"The response time of activity* invB *should not exceed 4 time units."*
   $\mathsf{G}(invB_{start} \rightarrow \mathsf{F}_{[0,4]}\, invB_{end})$

4. *"If activity* invB *has been invoked 4 times in the past 16 time units, than activity* recvR *will be executed within 32 time units."*
   $\mathsf{G}(\mathsf{C}^{16}_{=4} invB \rightarrow \mathsf{F}_{[0,32]} recvR)$

5. *"When activity* term *is executed, the average response time of all the invocations of activity* invB *completed in the past 720 time units should be less than 3 time units."*
   $\mathsf{G}(term_{end} \rightarrow \mathsf{D}^{720}_{\leq 3}(invB_{start}, invB_{end}))$

6. *"When activity* term *is executed, the average number of invocations, in an interval of 60 time units, of activity* invB *during the past 720 time units should be less than 4".*
   $\mathsf{G}(term_{end} \rightarrow \mathsf{V}^{720,60}_{\leq 4}(invB_{start}))$

7. *"When activity* term *is executed, the maximum number of invocations, in an interval of 60 time units, of activity* invB *during the past 720 time units should be less than 5".*
   $\mathsf{G}(term_{end} \rightarrow \mathsf{M}^{720,60}_{\leq 5}(invB_{start}))$

4.5 Translation to Linear Temporal Logic

In this section we show how SOLOIST can be translated into linear temporal logic. This translation guarantees the decidability of SOLOIST based on well-known results in temporal logic, allowing for its use with established verification techniques and tools. The translation presented here has not been designed to guarantee efficiency in verification but rather to be comprehensible.

SOLOIST is translated into a variant of linear temporal logic called MPLTL (Metric Linear Temporal Logic with Past) [125], which is a syntactically-sugared version of classical PLTL (see [86] and also section 2.2), defined over a mono-infinite discrete model of time represented by ω-words. For simplicity, we assume that the logic underlying SOLOIST is single-sorted; no expressiveness is lost, since it is well-known that many-sorted first-order logic (on which SOLOIST is based) can be reduced to single-sorted first-order logic when the number of sorts is finite. Moreover, since we assume that the domains corresponding to sorts are finite, we can drop the first-order quantification and convert each quantifier into a conjunction or a disjunction of atomic propositions. Similarly, n-ary predicate symbols (with $n \geq 1$) are converted into atomic propositions. For example, a formula of the form $\exists x \colon P(x)$, with x ranging over the finite domain $\{1, 2, 3\}$, is translated into the formula $\bigvee_{x \in \{1,2,3\}} P_x$, where P_1, P_2, P_3 are atomic propositions. We denote with Π the finite set of atomic propositions used in formulae obtained as described above.

These simplifications allow us to replace the temporal first-order structure $(\bar{\mathscr{D}}, \bar{\tau})$ and the variable assignment σ used in the definition of the satisfiability relation of SOLOIST with *timed* ω-words, i.e., ω-words over $2^{\Pi} \times \mathbb{N}$. For a timed ω-word $z = z_0, z_1, \ldots$, every element $z_k = (\sigma_k, \delta_k)$ contains the set σ_k of atomic propositions that are true at the natural timestamp denoted by $\tau_k = \sum_{i=0}^{k} \delta_i$ (with $\delta_i > 0$ for all $i > 0$). The satisfiability relation for SOLOIST can then be defined over timed ω-words, and it is denoted by $z, i \overset{\tau}{\models} \phi$, with z being a timed ω-word and $i \in \mathbb{N}$; we omit its definition since it can be derived with straightforward transformations from the one illustrated in figure 4.3.

Furthermore, we introduce a normal form where negations may only occur on atoms (see, for example, [125]). First, we extend the syntax of the language by introducing a dual version for each operator in the original syntax, except for the $\mathsf{C}^{K}_{\bowtie n}, \mathsf{V}^{K,h}_{\bowtie n}, \mathsf{M}^{K,h}_{\bowtie n}, \mathsf{D}^{K}_{\bowtie n}$ modalities[4]: the dual of $\wedge$ is $\vee$; the dual of U_I is "*Release*" R_I: $\phi \mathsf{R}_I \psi \equiv \neg(\neg\phi \mathsf{U}_I \neg\psi)$; the dual of S_I is "*Trigger*" T_I: $\phi \mathsf{T}_I \psi \equiv \neg(\neg\phi \mathsf{S}_I \neg\psi)$. For the sake of brevity, we do not explicitly report the semantics of these dual operators; it can be derived straightforwardly from the above definitions. A formula is in *positive normal form* if its alphabet is $\{\wedge, \vee, \mathsf{U}_I, \mathsf{R}_I, \mathsf{S}_I, \mathsf{T}_I, \mathsf{C}^{K}_{\bowtie n}, \mathsf{V}^{K,h}_{\bowtie n}, \mathsf{M}^{K,h}_{\bowtie n}, \mathsf{D}^{K}_{\bowtie n}\} \cup \Pi \cup \bar{\Pi}$, where $\bar{\Pi}$ is the set of formulae of the form $\neg p$ for $p \in \Pi$. For the rest of this section, we assume that SOLOIST formulae have been transformed into equivalent formulae in positive normal form.

Under these assumptions, the translation of SOLOIST to MPLTL boils down to expressing the temporal modalities $\mathsf{R}_I, \mathsf{T}_I, \mathsf{U}_I, \mathsf{S}_I, \mathsf{C}^{K}_{\bowtie n}, \mathsf{V}^{K,h}_{\bowtie n}, \mathsf{M}^{K,h}_{\bowtie n}, \mathsf{D}^{K}_{\bowtie n}$ in MPLTL, preserving their semantics.

First of all, we should remark that while in the semantics of SOLOIST the temporal information is denoted by a natural timestamp, in MPLTL the temporal information is implicitly defined by the integer position in an ω-word. However, the model based on timed ω-words and the one based on ω-words can be transformed into each other. Given an ω-word w such that $w, i \models \phi$ (where $w, i \models \phi$ denotes the satisfiability relation over ω-words), it is possible to define a timed ω-word $z = z_0, z_1, \ldots$, with $z_0 = (w_0, 0)$ and $z_k = (w_k, 1)$ for $k > 0$, such that $z, i \overset{\tau}{\models} \phi$. Conversely, given a SOLOIST timed ω-word z, we need to pinpoint in an MPLTL ω-word w the positions

[4]A negation in front of one of the $\mathsf{C}^{K}_{\bowtie n}, \mathsf{V}^{K,h}_{\bowtie n}, \mathsf{M}^{K,h}_{\bowtie n}, \mathsf{D}^{K}_{\bowtie n}$ modalities becomes a negation of the relation denoted by the $\bowtie$ symbol, hence no dual version is needed for them.

that correspond to timestamps in the z timed ω-word where an event occurred. We add to the set Π a special propositional symbol e, which is true in each position corresponding to a "valid" timestamp in the z timed ω-word. In the MPLTL semantics, an ω-word w over $\Pi \cup \{e\}$ is defined as follows: $w_k = \sigma_k \cup \{e\}$ whenever τ_k is defined, and $w_k = \emptyset$ otherwise. We then define a mapping ρ from SOLOIST dual normal form formulae into MPLTL formulae, such that we can state that $z, i \overset{\tau}{\models} \phi$ iff $w, \tau_i \models \rho(\phi)$. The mapping ρ is defined by induction as follows:

1. $\rho(p(t_1, \dots, t_n)) = p(t_1, \dots, t_n)$.
2. $\rho(\neg p(t_1, \dots, t_n)) = \neg p(t_1, \dots, t_n)$.
3. If ϕ and ψ are formulae and x is a variable, then

$$\begin{aligned}
\rho(\phi \wedge \psi) &= \rho(\phi) \wedge \rho(\psi);\\
\rho(\phi \vee \psi) &= \rho(\phi) \vee \rho(\psi);\\
\rho(\exists x \colon \phi) &= \exists x \colon \rho(\phi);\\
\rho(\forall x \colon \phi) &= \forall x \colon \rho(\phi).
\end{aligned}$$

4. If ϕ and ψ are formulae and I is a nonempty interval over $\mathbb{N}$, then

$$\begin{aligned}
\rho(\phi \mathsf{U}_I \psi) &= (\neg e \vee \rho(\phi))\mathsf{U}_I(e \wedge \rho(\psi));\\
\rho(\phi \mathsf{S}_I \psi) &= (\neg e \vee \rho(\phi))\mathsf{S}_I(e \wedge \rho(\psi));\\
\rho(\phi \mathsf{R}_I \psi) &= (e \wedge \rho(\phi))\mathsf{R}_I(\neg e \vee \rho(\psi));\\
\rho(\phi \mathsf{T}_I \psi) &= (e \wedge \rho(\phi))\mathsf{T}_I(\neg e \vee \rho(\psi)).
\end{aligned}$$

5. For $\mathsf{C}^K_{\bowtie n}$, we consider only the case $\mathsf{C}^K_{>n}$, since the other possible relations used for $\bowtie$ can be modeled with the following equivalences: $\mathsf{C}^K_{\leq n} \equiv \neg\mathsf{C}^K_{>n}$; $\mathsf{C}^K_{\geq n} \equiv \mathsf{C}^K_{>n-1}$; $\mathsf{C}^K_{<n} \equiv \neg\mathsf{C}^K_{>n-1}$; $\mathsf{C}^K_{=n} \equiv \mathsf{C}^K_{>n-1} \wedge \neg\mathsf{C}^K_{>n}$.

$$\rho(\mathsf{C}^K_{>n}(\phi)) = \bigvee_{0 \leq i_1 < \ldots < i_{n+1} < K} \left(\mathsf{Y}^{i_1}(e \wedge \phi) \wedge \ldots \wedge \mathsf{Y}^{i_{n+1}}(e \wedge \phi)\right)$$

where the MPLTL modality Y ("*yesterday*") is the past version of "*next*" and refers to the previous time instant. Intuitively, the above MPLTL formula states that in the previous K time instants there have been at least $n+1$ occurrences of the event corresponding to $(e \wedge \phi)$; such a situation satisfies the constraint associated with the original formula defined in SOLOIST.

6. The mapping for the $\mathsf{V}^{K,h}_{\bowtie n}$ modality is defined in terms of the C modality:

$$\rho(\mathsf{V}^{K,h}_{\bowtie n}\phi) = \rho(\mathsf{C}^{\lfloor \frac{K}{h} \rfloor \cdot h}_{\bowtie n \cdot \lfloor \frac{K}{h} \rfloor}\phi)$$

7. For the modality $\mathsf{M}^{K,h}_{\bowtie n}$, we include only the two cases $\mathsf{M}^{K,h}_{<n}$ and $\mathsf{M}^{K,h}_{>n}$, as the others can be derived by properly combining instances of these two:

$$\rho(\mathsf{M}^{K,h}_{<n}\phi) = \left(\bigwedge_{m=0}^{\lfloor \frac{K}{h} \rfloor - 1} \mathsf{Y}^{m \cdot h}\left(\rho\left(\mathsf{C}^h_{<n}\phi\right)\right)\right) \bigwedge \left(\mathsf{Y}^{\lfloor \frac{K}{h} \rfloor \cdot h}\left(\rho\left(\mathsf{C}^{(K \bmod h)}_{<n}\phi\right)\right)\right)$$

$$\rho(\mathsf{M}^{K,h}_{>n}\phi) = \left(\bigvee_{m=0}^{\lfloor \frac{K}{h} \rfloor - 1} \mathsf{Y}^{m \cdot h} \left(\rho \left(\mathsf{C}^{h}_{>n}\phi \right) \right) \right) \bigvee \left(\mathsf{Y}^{\lfloor \frac{K}{h} \rfloor \cdot h} \left(\rho \left(\mathsf{C}^{(K \bmod h)}_{>n}\phi \right) \right) \right)$$

The formulae above decompose the computation of the maximum number of occurrences of the event $(e \land \phi)$ by suitably combining constraints on the number of occurrences of the event in each observation interval within the time window.

8. For the $\mathsf{D}^{K}_{\bowtie n}$ modality, $\rho(\mathsf{D}^{K}_{\bowtie n}(\phi, \psi))$ is defined[5] as follows:

$$\bigvee_{0<h\leq\lfloor \frac{K}{2} \rfloor} \left(\bigvee_{\substack{0\leq i_1<j_1<\dots i_h<j_h<K \\ \text{and} \\ \left(\sum_{m=1}^{h} \frac{j_m-i_m}{h}\right) \bowtie n}} \left(\begin{array}{c} \mathsf{Y}^{i_1}(e \land \phi) \land \mathsf{Y}^{j_1}(e \land \psi) \land \\ \dots \\ \land \mathsf{Y}^{i_h}(e \land \phi) \land \mathsf{Y}^{j_h}(e \land \psi) \land \\ \neg \left(\bigvee_{\substack{0\leq s<t<K \\ s\notin\{i_1,\dots,i_h\} \\ t\notin\{j_1,\dots,j_h\}}} \left(\mathsf{Y}^{s}(e \land \phi) \land \mathsf{Y}^{t}(e \land \psi) \right) \right) \end{array} \right) \right)$$

The above formula considers all possible h occurrences (with h up to $\lfloor \frac{K}{2} \rfloor$, as indicated in the outer "or") of pairs of events corresponding to $(e \land \phi)$ and $(e \land \psi)$. The inner "or" considers a sequence of h pairs of time instants $(i_1, j_1), \dots (i_h, j_h)$, constrained by the bound represented by $\bowtie n$. The top, right-hand part of the formula imposes that every pair of time instants actually corresponds to the occurrence of a pair of events; the bottom, right-hand part excludes the case that some pairs of events may occur at time instants which are not in the above sequence.

The complexity of a formula resulting from the translation may be exponential in the size of the constants occurring in the aggregate operators. Without aggregate operators, the translation is linear in the size of the original formula. The only relevant cases for aggregate operators are $\mathsf{C}^{K}_{>n}$ and $\mathsf{D}^{K}_{\bowtie n}$, since the other modalities can easily be defined in terms of these two. The mapping for $\mathsf{C}^{K}_{>n}\phi$ considers all subsets of $n+1$ integers of the set $\{0, \dots, K-1\}$. Hence, it may require an MPLTL formula of size proportional to $(n+1)\binom{K}{n+1}$, which in the worst case, corresponding to $n+1 = \frac{K}{2}$, is $O(K \cdot 2^K)$. The mapping of $\mathsf{D}^{K}_{\bowtie n}(\phi, \psi)$ essentially requires, in the worst case, to select all possible subsets of set $\{0, \dots, K-1\}$, i.e., 2^K subsets. Hence, again this may require an MPLTL formula of size $O(K \cdot 2^K)$. As remarked at the beginning of this section, the translation presented above has been designed to show the possibility of reducing SOLOIST to a linear temporal logic; nevertheless, future work will address efficiency in the verification of SOLOIST formulae.

[5]For the sake of simplicity, we consider the case of only one pair of events (ϕ, ψ), but the formula can be generalized to the case of multiple pairs (ϕ_i, ψ_i).

4.6 Summary

This chapter introduced SOLOIST, a specification language for service compositions interactions. The language is based on a many-sorted first-order metric temporal logic, which has been extended with new temporal modalities that support aggregate operators for events occurring in a certain time window. Expressiveness was not the sole requirement in designing this language. We also wanted the language to express specifications that could lead to automatic formal verification. Indeed, we also show that SOLOIST, under certain assumptions, can be translated into linear temporal logic, allowing for its use with established techniques and tools, both for design-time and for run-time verification.

Chapter 5

Intermezzo 1: Specification - State of the Art

In this chapter we report on the state of the art of specification languages for SBAs (section 5.1) and of property specification patterns (section 5.2).

5.1 On Specification Languages for SBAs

During the field study described in chapter 3, we noticed that the three main formal languages used by researchers in the field of SBAs to specify and verify properties related to service interactions are LTL (Linear Temporal Logic), CTL (Computational Tree Logic), and Event Calculus [91]. While the first two are mainly used to describe untimed temporal relations between events, Event Calculus has been the basis to develop more expressive languages, such as EC-Assertion [108], which can express service guarantees terms such as those captured by patterns S1 and S2. However, it requires to introduce additional constructs in a formula, such as explicit variables to track response time or event counters, as well as additional support formulae, like the ones used to maintain a list of variables that are used to compute an aggregate value. These additional variables and formulae decrease the readability of specifications and make writing them more cumbersome and error-prone. We also noticed a recurring presence of extensions of temporal logics with support for first-order quantification, namely LTL-FO, CTL-FO [54], LTL-FO+ [75], and CTL-FO+ [76], which enrich the underlying logic to express *data-aware* properties, captured by pattern S7.

Other specification languages, like WS-CoL [12] and RTML [6], are proposed as assertion languages for BPEL compositions to promote "design by contract" [113], and are usually integrated in a dynamic monitoring architecture. They are reminiscent of assertion languages that were designed for specific programming languages such as ANNA [106], an annotation language for Ada, and JML [99], the Java Modeling Language.

In the realm of SBAs there have also been several proposals of languages for specifying service level agreements, mainly targeting QoS attributes such as response time and throughput; among them, we mention WSLA [87] and a timeliness-related extension of WS-Agreement [115]. These languages usually do not have any formal or mathematical grounding, but in most cases they define an XML schema containing the definition of the main QoS attributes and their data types. One exception is SLAng, which—besides being defined on the top of standard modeling languages like EMOF and OCL, to guarantee precision and understandability—has been mapped to timed automata, to enable efficient run-time monitoring [129].

The fragment of SOLOIST corresponding to many-sorted metric first-order temporal logic is very similar to the work defined in [14], where a similar fragment is used to define system policies, which are then monitored; however, this fragment, without the other temporal operators introduced in SOLOIST, would have been inadequate to express all the service provisioning patterns identified in our field study.

In the field of (temporal) logics, there have been several proposals to express properties related or similar to the ones captured by the service provisioning patterns described in chapter 3. For example, references [96] and [97] propose, respectively, Counting CTL and Counting LTL, which extend the temporal modalities of the underlying (non-metric) logic with the ability to constrain the number of states satisfying certain sub-formulae along paths. In [16], a first-order policy specification language is introduced; the language, based on past time linear temporal logic with first-order quantifier, includes also a counting quantifier, used to express that a policy depends on the number of times another policy was satisfied in the past. Rabinovich [128] presents TLC, the metric temporal logic with counting modalities over continuous time, where a counting modality $C_k(X)$ states that X is true at least at k points in the unit interval ahead.

Aggregate operators have been studied in the context of mathematical logic, for database query languages [77] and logic programming [122]. More recently, they have also been considered in temporal logics, to express quantitative atomic assertions related to accumulative values of variables along a computation [42]. de Alfaro [53] introduces an operator to express bounds on the average time between events (conceptually similar to the D operator of SOLOIST) in the context of probabilistic temporal logic, to specify and verify performance and reliability properties of discrete-time probabilistic systems. Extensions of specification formalism with statistical operators have also been proposed in the context of run-time verification. In [60], LTL is extended with operators that evaluate aggregate statistics over an execution trace. Reference [68] presents the LARVA verification tool, based on *Dynamic Automata with Timers and Events*, which is able to evaluate statistical measures over dynamic intervals, like the ones identified with the $\mathsf{C}, \mathsf{V}, \mathsf{M}, \mathsf{D}$ modalities of SOLOIST; however, the report does not provide enough details on the language used to specify the properties to monitor.

5.2 On Property Specifications Patterns

Although in the study described in chapter 3 we have considered only three systems of specification patterns (plus the “service provisioning” one derived from the study itself), other similar systems have been presented in the literature. Below, we briefly summarize the pattern systems we did not consider for the study and explain why we opted for the ones presented in section 3.2.

In the area of qualitative temporal specifications, a catalogue of safety patterns is presented in [39]; however, with respect to the “D” group, it is restricted only to safety patterns occurring in the specification of industrial automation systems. Other extensions of the “D” patterns are proposed in [45], which deals with the support of events in LTL formulae, and in [137], where the PROPEL approach—based on a “disciplined” natural language and finite state automata—is used to express fine-tuned versions of the “D” patterns. Since in our case studies we wanted to assess the usage of the “D” patterns at a high level, we did not go for such more specialized versions of these patterns. As for the area of real-time specifications, a system of patterns using structured English sentences is described in [62]; however, this work is tailored for clocked computational tree logic, while we wanted to use specification language-agnostic pattern systems, such as the “R” and “G” groups. VTS (Visual Timed Event Scenario) [3] is a visual pattern language for expressing complex relations between timed events, supporting real-time constraints.

Another class of specification patterns we did not include in the study is represented by patterns for probabilistic quality properties [74]. For the sake of completeness, we should say that three requirements specifications from the set of research case studies were actual matches for two of the patterns introduced in [74], while none of the specifications of the set of industrial case studies could be expressed using a probabilistic property pattern. Patterns for probabilistic satisfaction of quantitative properties are described in [101].

The study described in chapter 3 is also one of the few that reports quantitative data on the usage of certain specification pattern systems in practical examples. Similar data can also be found in [57], as shown in section 3.3.1; in [90], though the usage distribution of each pattern is not actually disclosed; in [74], for probabilistic property patterns; in [124], which presents a study—conceptually similar to ours—on the analysis of the usage of the “R” patterns in the automotive domain.

The “D” group is also at the base of some work that focuses on the specification and verification of service interactions in SBAs. For example, in [102], property patterns are defined in an ontology, whose concepts can then be used by developers to describe the interaction behavior of services as constraints. These constraints specify the occurrence and sequencing rules of service invocations and are checked at run time by a dedicated monitoring infrastructure. A similar approach is also followed in [135], where service conversations are specified using a subset of UML 2.0 Sequence Diagrams, which are shown to be able to express all the "D" patterns. Reference [149] presents PROPOLS, a specification language based on the "D" patterns, which adds support for the logic composition of patterns; this language can be used to describe some properties against which service composition workflows can be checked for compliance with a static verification tool. Another specification language, PL, also based on the "D" group, is presented in [144]; the language is used to express behavioral properties of business processes, which can then be automatically translated into a process algebra for refinement checking.

Other work has defined specification languages for service interactions based on real-time patterns, as for the case of the XTUS-Automata language proposed in [85], which also presents the companion run-time monitoring infrastructure. This work presents two additional patterns, "temporal properties over cardinalities" and "absolute time properties", which match, respectively, the S2 and S5 patterns identified in our study.

Part III

Verification

Chapter 6

Interface Decomposition for Service Compositions

6.1 Overview

In the dynamic and evolvable settings that characterize (service-based) open-world software, service providers, in general, make available to service integrators only the syntactical interface of the services they provide. It is often unrealistic to assume that service providers will also make available some sort of "richer" interface descriptions (e.g., a behavioral specification) of their services. This happens despite the fact that such "richer" interface descriptions could be used by service integrators to assess that a certain external service they rely on can contribute to fulfill the functional requirements specifications of their composite applications.

It is then clear that an automated technique for deriving, from the requirements specification of a composite service, the required interface of its partner services, could improve the process followed by service integrators to assemble service compositions.

In this chapter, we propose a technique for the automatic generation of the behavioral interfaces of the partner services, by *decomposing* the requirements specification of a service composition. Our technique generates behavioral interfaces that constitute required specifications for the partner services; these specifications guarantee that the composite service will fulfill its required safety properties at run time, while it interacts with the external services. Since we assume that the behavioral descriptions of external services are not available, our technique is based on the purely syntactical knowledge of their interfaces.

Once the behavioral specifications of the external services have been inferred, they can serve multiple purposes. For example, they can be used with (semi-)automatic composition mechanisms, for selecting the services that fulfill in the best way the functional requirements of the composite service. Moreover, they can become clauses of the SLAs negotiated with service providers. Furthermore, they can be translated into verifiable run-time properties, which can be monitored while the system is operating, to check if the external services behave as expected, e.g., to check if the service providers meet the obligations they signed in the SLAs.

We use LTS (see section 2.3) to model the behavior of service compositions, the global specifications of the environment with which a composite service interacts, and the behavioral interfaces of the individual services.

The chapter is structured as follows. After describing the running example (section 6.2), we introduce our formal models for service compositions and their interface specification in section 6.3. Section 6.4 presents the interface decomposition problem, illustrates our technique to solve it, and shows its correctness. Section 6.5 discusses some approaches for the validation of the decomposition technique, as well as its shortcomings. Section 6.6 reports on the application of our approach to two case studies.

6.2 Running Example

Our running example is a simplified version of the *Car Rental Agency* one presented in [32]; we call it *Simple Car Rental (SCR)*. The example illustrates a service composition that is run at a car rental office branch. The composite service interacts with a *Car Broker (CB)* service, which controls the operations of the branch; with a *User Interaction (UI)* service, through which customers can make car rental requests; with a *Car Information (CI)* service, which maintains a database of cars availability and allocates cars to customers; with a *Car Parking Sensor (CPS)* service, which exposes as a service the sensor that senses cars as they are driven in or out of the parking lot of the branch. The workflow of the composite service is sketched in figure 6.1.

The *SCR* service starts when it receives the `startRental` message from the *CB* service. It then enters an infinite loop; at each iteration it can receive one of the following messages:

- `findCar`. A customer requests to rent a car; the *SCR* service checks the availability of a car by invoking the `lookupCar` operation on the *CI* service. The `lookupCar` operation returns its result—which can be either a negative answer or an identifier corresponding to the digital key to access the car—in the `result` variable, which is then passed as parameter to the `findCarCB` operation, a callback invoked on the *UI* service.

- `carEnter` and `carExit`. These two messages are sent out by the *CPS* service when a car enters (respectively, exits) the parking lot. The process reacts to this information by updating the cars database, invoking, respectively, the `markAvailable` and `markUnavailable` operations on the *CI* service.

- `stopRental`. The *CB* service stops the operations of the branch, terminating also the composite service.

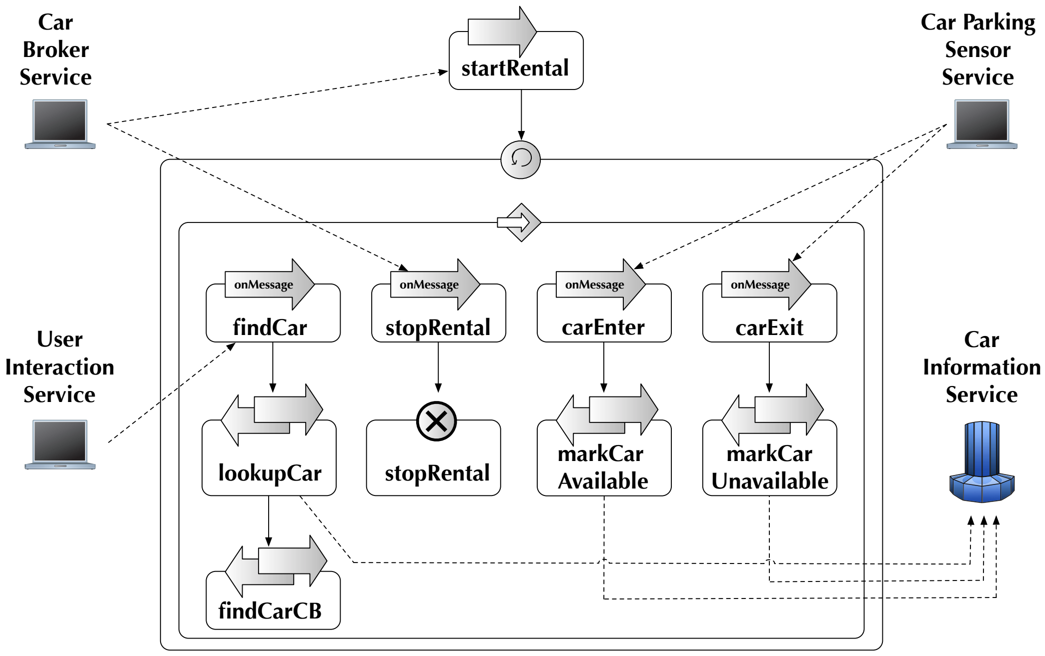


Figure 6.1. **The *Simple Car Rental* example**

To keep the example compact, we assume that a single car is available in the branch, and that the *CI* service is accessed only by the *SCR* service instance running in the branch.

The correct execution of the *SCR* service depends on the functionalities provided by the *CI* and *CPS* services. Therefore, in the next two sections we show the application of our interface decomposition technique to derive the behavioral interfaces of these two services.

6.3 Service Composition and Global Interface Specification Models

In this section we present the formal model of service compositions and describe how we can infer the global interface specification of the environment (i.e., the set of partner services) with which a composite service interacts. We refer the reader to figure 6.2, for mapping symbols onto components.

6.3.1 Service Composition

A service composition C interacts with a set of external services denoted as $E = \{E_1, \ldots, E_n\}$. Each service $E_i \in E$ makes available a set of operations $O^i = \{o^i_1, \ldots, o^i_m\}$, representing its syntactical interface. We assume that $\forall i, j, 1 \leq i \leq n, i < j \leq n, O^i \cap$

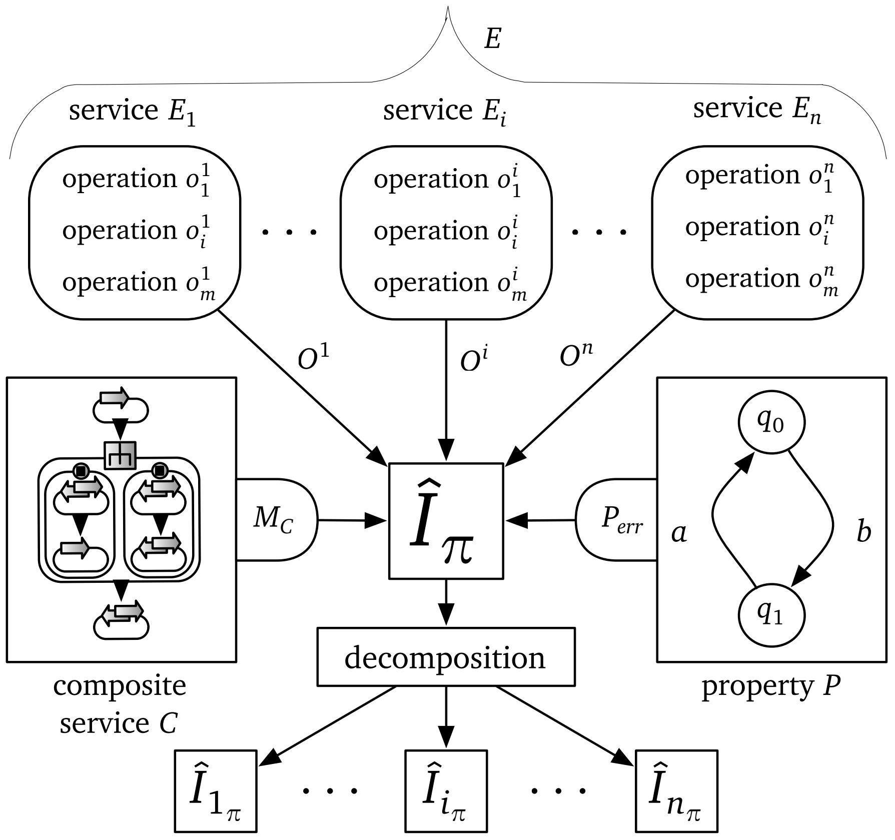


Figure 6.2. **Notation and general model of the service interface decomposition problem**

$O^j = \varnothing$, since each operation can be unambiguously identified by its name combined with the name of the service it belongs to (e.g., by means of the interface and service elements of a WSDL 2.0 description [143]).

We assume that service compositions are implemented as BPEL processes, which can be formalized in terms of labeled transition systems as shown in [65], with tools such as WS-Engineer [64]. For a service C, let M_C be the corresponding LTS.

The safety requirements on the behavior of the composite service C, when it interacts with the external services E, can be modeled by a property LTS P. This LTS can be synthesized, for example, from a specification in a temporal logic formalism such as LTL or Fluent LTL [71]. Note that the property P implicitly defines the unwanted behaviors, by means of the corresponding error LTS P_{err}.

Modeling the *SCR* Example

In the example, we are interested in the environment constituted by the services *CI* and *CPS*, so we have $E = \{CI, CPS\}$, $O^{CI} = \{$`markAvailable`, `markUnavailable`, `lookupCar`$\}$ and $O^{CPS} = \{$`carEnter`, `carExit`$\}$.

In the rest of this chapter, we use the FSP textual notation [107] to compactly represent LTS models. In FSP, identifiers beginning with a lowercase letter denote actions while identifiers beginning with an uppercase letter denote processes (states in the underlying LTS); the symbol "`->`" denotes the action prefix operator, while the vertical bar "`|`" denotes the choice operator. The following code snippet corresponds to the LTS model of the *SCR* service:

```
range KEY = 0..1 //(0 means car not available)
SCR  = (startRental -> Main),
Main = (findCar -> lookupCar[result:KEY] -> findCarCB[result] -> Main
        | carExit -> markUnavailable -> Main
        | carEnter -> markAvailable -> Main
        | stopRental -> END).
```

Note that each operation invoked on the *SCR* service and on its partner services is modeled as an action. Moreover, since the variable `result` ranges over the domain `KEY`, the `lookupCar` action is internally represented as `lookupCar[0]` and `lookupCar[1]`; the same applies to `findCarCB`.

Service Behavior

The expected behavior of the *SCR* service is expressed by the following requirement:

> *"If the car enters the parking lot, and it does not exit until a customer requests it for renting, then this request should not return a negative answer."*

This requirement can be formalized in Fluent LTL as the formula $\mathsf{G}(CarIn \Rightarrow \psi)$, where *CarIn* is a fluent that changes value when the car is in the parking lot, and it is defined as $CarIn \equiv \langle$`carEnter`, `carExit`$\rangle$ *initially False*; ψ is the auxiliary formula `findCar` $\Rightarrow$ ($\neg$`findCarCB[0]` W `findCarCB[1]`). Here G and W are, respectively, the LTL temporal operators "globally" and "weak until". This Fluent LTL formula represents a safety property and thus can be translated automatically [71] into an (error) LTS model, whose textual description is shown below:

```
Perr  = Q0,
   Q0 = ({carExit, findCar, findCarCB[0..1]} -> Q0
         |carEnter -> Q1),
   Q1 = (carExit -> Q0
         |{carEnter, findCarCB[0..1]} -> Q1
         |findCar -> Q2),
   Q2 = (findCarCB[0] -> ERROR
         |findCarCB[1] -> Q1
         |{carEnter, findCar} -> Q2
         |carExit -> Q3),
   Q3 = (findCarCB[0] -> ERROR
         |findCarCB[1] -> Q0
         |carEnter -> Q2
         |{carExit, findCar} -> Q3).
```

6.3.2 Global Interface Specification

The first step for defining the interface decomposition technique is to characterize the global expectations from E in order for C to fulfill its requirement P; i.e., we want to infer the global interface specification of the environment E with which C interacts. By following the technique introduced in [72] and summarized below, we can determine the global interface specification by computing the LTS $\hat{I}_\pi$, with $\hat{I}_\pi = \textsc{BuildInterface}((M_C \parallel P_{err}), O)$, where $O = \bigcup_{i=1}^{|E|} O^i$.

The pseudo-code of function BUILDINTERFACE is shown in figure 6.3. The function receives as first parameter an LTS *model*; the actual parameter that is passed ($M_C \parallel P_{err}$) contains all the traces that violate the property P. The actual value of the second parameter *actions*, is the set of all the operations provided by the external services and is used on line 2 as an operand of the *interface* operator, to get the LTS named *gen_interface*. This LTS is further processed with a special determinization step (line 3), provided internally by the LTSA tool [107]. This determinization step performs τ-elimination and subset construction but, unlike standard automata theory algorithms, it handles in a special way the π state. Since during the subset construction the states of the deterministic LTS correspond to *set of states* of the original, non-deterministic LTS, if any of the states in the set is π, then the entire set becomes a π state in the deterministic LTS. This means that a trace that non-deterministically may or may not lead to the error state has to be considered an error trace. In practical terms, it means that performing a certain sequence of actions on the external services does not guarantee that the service composition will not reach an error state. Subsequently, the LTS *gen_interface* is completed (line 4) with a sink state and the transitions

```
1: function BuildInterface(model, actions)
2: gen_interface ← model ↑ actions
3: Determinize(gen_interface)
4: CompleteWithSinkState(gen_interface)
5: return gen_interface
```

Figure 6.3. **Pseudo-code of the BUILDINTERFACE function**

leading to it, by invoking an auxiliary function. The missing transitions in the original LTS represent behaviors of the external services that are never exercised by the service composition; with the completion, they are made sink behaviors and thus no restriction is imposed on them.

The notation used for the resulting LTS, $\hat{I}_\pi$, denotes that it contains the error state (deriving from the error LTS P_{err}) and that it has been completed with a sink state. Hereafter, we use the notation $\hat{I}$ to refer to the variant of $\hat{I}_\pi$ that does not contain the error state, without the transitions leading to it. In symbols, given $\hat{I}_\pi = \langle Q \cup \{\pi\}, \alpha\hat{I}_\pi, \delta, q_0\rangle$, $\hat{I} = \langle Q, \alpha\hat{I}, \delta', q_0\rangle$, where $\alpha\hat{I} = \alpha\hat{I}_\pi$, $\delta' = \delta \setminus \{(q, a, \pi) \mid a \in \alpha\hat{I}_\pi\}$.

Application to the Example

The first parameter passed to the BUILDINTERFACE function is `(SCR || Perr)`. As for the second parameter, the list of actions passed to the function is composed by `markAvailable`, `markUnavailable`, `lookupCar[0]` and `lookupCar[1]` (from *CI*), and by `carEnter` and `carExit` (from *CPS*). The resulting interface $\hat{I}_\pi$ is defined as follows:

```
Ipi  = Q0,
Q0   = (lookupCar[0..1] -> Q0
        |carExit -> Q1
        |carEnter -> Q2
        |{markUnavailable, markAvailable} -> SINK),
Q1   = (markUnavailable -> Q0
        |{carExit, carEnter, markAvailable, lookupCar[KEY]} -> SINK),
Q2   = (markAvailable -> Q3
        |{carExit, carEnter, markUnavailable, lookupCar[KEY]} -> SINK),
Q3   = (lookupCar[0] -> ERROR
        |carExit -> Q1
        |carEnter -> Q2
        |lookupCar[1] -> Q3
        |{markUnavailable, markAvailable} -> SINK),
SINK = ({carExit, carEnter, markUnavailable, markAvailable,
         lookupCar[KEY]} -> SINK).
```

6.4 Decomposing Interface Specifications

The method described in section 6.3.2 computes the global interface specification of a service composition, i.e., the behavior that its partner services, *considered as a whole*, should manifest in order for the composite service to fulfill its requirements specification. However, this "centralized" solution is not realistic for the domain of SBAs, since each service is operated independently by its own provider, and has no knowledge of the other services with which its client service (i.e., a composite service) interacts.

Therefore, we argue it is necessary to define a more "distributed" approach, which generates the individual behavioral interfaces for the partner services.

To this end, we define the interface decomposition problem as follows (refer to figure 6.2 for mapping symbols onto components): given a service composition C, which interacts with a set of external services $E = \{E_1, \ldots, E_n\}$ whose most general or permissive behavior, as a whole, is represented by I, we decompose I into interface specifications for the individual partner services, denoted as $I_i, 1 \leq i \leq |E|$.

The individual interface specifications obtained by means of the interface decomposition technique should guarantee that the composite service fulfills its requirement specification. This correctness requirement can be formally stated as:

$$\left(\mathop{\Big\|}_{i=1}^{|E|} I_i \right) \| M_C \models P$$

In the rest of this section, we illustrate our technique for decomposing interface specifications and show its application to the *SCR* example. We first present a basic approach to the problem and observe that it generates over-constraining interfaces. Subsequently, we propose our heuristic-based technique, which generates less constraining, but still correct behavioral interfaces.

Since the correct execution of the *SCR* example depends on the functionalities provided by the *CI* and *CPS* services, in the next two subsections we use our interface decomposition technique to derive the behavioral interfaces of these two services.

6.4.1 Basic Decomposition Approach

A first approach to the problem of interface decomposition can be based on the intuition that each external service can contribute to the global interface specification only through the operations that it provides. Formally, this means the interface specification $\hat{I}_{i_\pi}$ of an external service E_i can be computed as $\hat{I}_{i_\pi} = \textsc{BuildInterface}(\hat{I}_\pi, O^i)$.

Note that $\hat{I}_{i_\pi}$ contains the error state; as done for the case of the global interface specification, we use the notation $\hat{I}_i$ to refer to the variant of $\hat{I}_{i_\pi}$ that contains neither the error state nor the transitions leading to it.

However, simple experimentation with this technique reveals that such an approach generates interfaces that are too restrictive. For example, its application to the running example generates the following interface specifications.

For the *CI* service, we restrict the global interface specification over the alphabet {`markUnavailable`, `markAvailable`, `lookupCar[0]`, `lookupCar[1]`}. The resulting LTS is:

```
CI = Q0,
Q0 = ({lookupCar[0..1], markUnavailable} -> Q0
        |markAvailable -> Q1),
Q1 = (lookupCar[0] -> ERROR
        |markUnavailable -> Q0
        |{lookupCar[1], markAvailable} -> Q1).
```

It states that after a `markAvailable` operation, when the computation is in state `Q1`, the `lookupCar` operation will return successfully (i.e., a value different from 0). Essentially, state `Q1` denotes the fact that the car is in the parking lot.

As for the *CPS* service, the global interface specification is restricted over the alphabet {`carEnter`, `carExit`}. The resulting LTS is:

```
CPS  = Q0,
 Q0  = (carEnter -> ERROR
        |carExit -> Q0).
```

This interface is too restrictive, since it disallows a car from ever entering the parking lot. Furthermore, considering that according to the definition of the fluent *CarIn*, the car is initially out of the parking, this interface in practice blocks any behavior from the car.

In fact, we can make a stronger observation about the individual interfaces built in this way:

Proposition 1. *Let* $\hat{I}_{i_\pi} = \textsc{BuildInterface}(\hat{I}_\pi, O^i)$*, and* $\hat{I}'_{i_\pi} = \textsc{BuildInterface}((M_C \parallel P_{err}), O^i)$*. Then* $\hat{I}_{i_\pi}$ *and* $\hat{I}'_{i_\pi}$ *are isomorphic.*

Proof. By construction, function BuildInterface generates a canonical deterministic LTS whose error traces are equal to the error traces of its first argument projected to the alphabet represented by its second argument [72]. Since $\hat{I}_\pi = \textsc{BuildInterface}((M_C \parallel P_{err}), O)$, it follows that $errTr(\hat{I}_\pi) = errTr((M_C \parallel P_{err}) \uparrow O)$. In a similar way, $errTr(\hat{I}_{i_\pi}) = errTr(\hat{I}_\pi \uparrow O^i)$. From these two statements, we derive that $errTr(\hat{I}_{i_\pi}) = errTr(((M_C \parallel P_{err}) \uparrow O) \uparrow O^i)$. Since $O_i \subseteq O$, we conclude that $errTr(\hat{I}_{i_\pi}) = errTr((M_C \parallel P_{err}) \uparrow O^i)$. Additionally, $\hat{I}'_{i_\pi} = \textsc{BuildInterface}((M_C \parallel P_{err}), O^i)$ implies that $errTr(\hat{I}'_{i_\pi}) = errTr((M_C \parallel P_{err}) \uparrow O^i)$. Since the error traces of $\hat{I}_{i_\pi}$ and $\hat{I}'_{i_\pi}$ are equal, we conclude that the canonical representations $\hat{I}_{i_\pi}$ and $\hat{I}'_{i_\pi}$, generated by function BuildInterface, are isomorphic, and therefore so are $\hat{I}_i$ and $\hat{I}_i{}'$. □

As a result, each interface that we compute in this fashion is sufficient by itself, to guarantee the global property on the system, meaning that $\forall i, (\hat{I}_i \parallel M_C) \models P$, which implies that $\left(\parallel_{i=1}^{|E|} \hat{I}_i\right) \parallel M_C \models P$.

However, imposing such interfaces would be overly constraining. Moreover, a solution that assigns the entire responsibility for achieving the global property to every single service is not desirable. Ideally, we would like a solution that distributes the responsibility to the partner services in a way that allows as much participation from each service as possible in the behavior of the service composition. To this end, in the next section we propose a heuristic that avoids to unnecessarily constrain the interface of partner services that cannot lead to error behaviors of the system.

6.4.2 Heuristic-based Decomposition Technique

The heuristic we propose to use is based on inspecting the actions that label the transitions that lead to the error state in the global interface specification. It may be the case that none of these actions corresponds to one of the operations provided by the partner service (hereafter referred to as E_i) for which we want to compute the behavioral interface. This means that service E_i will never cause an error behavior in the system constituted by the composite service and its partner services. In this case, the behavioral interface of E_i can be obtained by decomposing a simplified model of the global interface specification, which does not include the error behaviors that are not directly ascribable to E_i.

More formally, for a service E_i with actions O^i, given a global interface specification $\hat{I}_\pi = \langle Q, \alpha I, \delta, q_o \rangle$, the heuristic builds an auxiliary global interface specification, denoted with $I_{heu}(i)$. This heuristic-based, auxiliary interface specification is computed as $I_{heu}(i) = \langle Q, \alpha I, \delta', q_o \rangle$, where $\delta' = \delta \setminus \{(q, a, \pi) \mid a \notin O^i\}$. The definition of δ' shows that the heuristic removes the transitions to the error state labeled with actions (operations) not provided by E_i. Note that as a result of removing such transitions, $I_{heu}(i)$ may not be complete; note also the error state may be removed in case the error transitions were ascribable only to the other services different from E_i. The interface specification of the service E_i, denoted with $\hat{I}_{i_\pi}$, can then be computed as $\hat{I}_{i_\pi} = \textsc{BuildInterface}(I_{heu}(i), O^i)$.

Correctness

Before showing that this technique is a correct solution of the interface decomposition problem, we introduce and prove some helper propositions.

Proposition 2. *Given $\hat{I}_\pi$ and $\hat{I}_{i_\pi}$ defined as above, the relation $errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \supseteq errTr(\hat{I}_\pi)$ holds.*

Proof. The proof is by contradiction. Suppose there is a trace t, such that $t \in errTr(\hat{I}_\pi)$ and that $t \notin errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi})$. Let a be the last action in t, and (q, a, q') the corresponding transition that leads to the error state in $\hat{I}_\pi$. Since there must exist a k such that $a \in O^k$, we know that transition (q, a, q') will not be removed from $I_{heu}(k)$. From the semantics of the interface operator, we can then conclude that $(t \restriction O^k) \in errTr(\hat{I}_{k_\pi})$. Since for all i, $\hat{I}_{i_\pi}$ is complete, we also know that $t \in Tr(||_{i=1}^{|E|} \hat{I}_{i_\pi})$. But since t leads to the error state with at least one component of this, we conclude that $t \in errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi})$, which is a contradiction. □

Proposition 3. *Given $\hat{I}$ and $\hat{I}_i$ defined as above, the relation $Tr(||_{i=1}^{|E|} \hat{I}_i) \subseteq Tr(\hat{I})$ holds.*

Proof. Consider the set O of all the operations made available by the external services; let O^* represent its Kleene closure. Similarly, let O^{i^*} be the Kleene closure of the set of operations provided by an individual external service E_i. By construction, $\hat{I}$ is obtained from $\hat{I}_\pi$ by removing the error state and the transitions leading to it. Hence, since no trace of the $\hat{I}$ interface leads to the error state, we know that $Tr(\hat{I}) = O^* \setminus errTr(\hat{I}_\pi)$; similarly, $\forall i, 1 \leq i \leq |E|, Tr(\hat{I}_i) = O^{i^*} \setminus errTr(\hat{I}_{i_\pi})$. Moreover, we know that a composite process has an error trace, if at least one of its constituent processes has an error trace. In symbols:

$$errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) = \Big\{ t \in Tr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \mid (t \upharpoonright O^1) \in errTr(\hat{I}_{1_\pi}) \\ \vee (t \upharpoonright O^2) \in errTr(\hat{I}_{2_\pi}) \vee \cdots \vee (t \upharpoonright O^{|E|}) \in errTr(\hat{I}_{|E|_\pi}) \Big\}.$$

Hence:

$$O^* \setminus errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) = \Big\{ t \in Tr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \mid \\ (t \upharpoonright O^1) \notin errTr(\hat{I}_{1_\pi}) \wedge \cdots \wedge (t \upharpoonright O^{|E|}) \notin errTr(\hat{I}_{|E|_\pi}) \Big\}$$

$$= \Big\{ t \in Tr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \mid (t \upharpoonright O^1) \in \Big(O^{1^*} \setminus errTr(\hat{I}_{1_\pi})\Big) \wedge \cdots \wedge (t \upharpoonright O^{|E|}) \in \Big(O^{|E|^*} \setminus errTr(\hat{I}_{|E|_\pi})\Big) \Big\}$$

$$= \Big\{ t \in Tr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \mid (t \upharpoonright O^1) \in Tr(\hat{I_1}) \wedge \cdots (t \upharpoonright O^{|E|}) \in Tr(\hat{I_{|E|}}) \Big\} = Tr(\overset{|E|}{\underset{i=1}{||}} \hat{I}_i).$$

Since $errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \supseteq errTr(\hat{I}_\pi)$ holds from Proposition 2, $O^* \setminus errTr(||_{i=1}^{|E|} \hat{I}_{i_\pi}) \subseteq O^* \setminus errTr(\hat{I}_\pi)$ also holds. Hence $Tr(||_{i=1}^{|E|} \hat{I}_i) \subseteq Tr(\hat{I})$. □

We can now show the correctness of our heuristic-based decomposition technique, by stating and proving the following proposition.

Proposition 4 (Correctness)**.** *Given the model of a service composition M_C and the specification of its desired behavior P when interacting with a set of external services E, the interfaces of the individual external services $\hat{I}_i, 1 \leq i \leq |E|$, when computed applying the aforementioned heuristic, satisfy the following relation: $\left(||_{i=1}^{|E|} \hat{I}_i\right) \,||\, M_C \models P$.*

Proof. From [72], we know $\hat{I} \,||\, M_C \models P$. Furthermore, from Proposition 3, $\left(||_{i=1}^{|E|} \hat{I}_i\right) \models \hat{I}$. It follows that $\left(||_{i=1}^{|E|} \hat{I}_i\right) \,||\, M_C \models P$. □

Application to the Example

By analyzing the global interface specification `Ipi` showed in section 6.3.2, we notice that the error state can be reached by executing, in state `Q3`, the transition labeled with lookupCar[0], which is an operation provided by the *CI* service. The heuristic described above can then be applied to compute the interface for the *CPS* service.

We first create a refined model of the global interface, by removing the transitions that lead to the error state and that are not labeled with actions belonging to the alphabet of the *CPS* service:

```
Ipi_cps  = Q0,
Q0       = (lookupCar[0..1] -> Q0
            |carExit -> Q1
            |carEnter -> Q2
            |{markUnavailable, markAvailable} -> SINK),
Q1       = (markUnavailable -> Q0
            |{carExit, carEnter, markAvailable, lookupCar[KEY]} -> SINK),
Q2       = (markAvailable -> Q3
            |{carExit, carEnter, markUnavailable, lookupCar[KEY]} -> SINK),
Q3       = (carExit -> Q1
            |carEnter -> Q2
            |lookupCar[1] -> Q3
            |{markUnavailable, markAvailable} -> SINK),
SINK     = ({carExit, carEnter,
             markUnavailable, markAvailable, lookupCar[KEY]} -> SINK).
```

Next, the global interface specification is restricted over the alphabet {carEnter, carExit}; the resulting LTS is:

```
CPS  = Q0,
 Q0  = ({carEnter, carExit} -> Q0).
```

As expected, this new interface, obtained for the *CPS* service with the application of the heuristic, allows for more behaviors than the one computed with the basic technique. More specifically, in this case the interface represents the *universal interface* of service *CPS*, i.e., the interface that allows any of its operations. Since the error behaviors of the system are prevented by the interface of the other service (*CI*), there is no need to constrain the interface of *CPS*.

As for the interface specification of service *CI*, the application of the heuristic does not affect its generation, i.e., it coincides with the one shown in section 6.4.1.

6.5 Discussion

6.5.1 Validation of the Generated Interfaces

Although the definition of the interface decomposition problem includes a correctness requirement, which guarantees that the generated interfaces will not lead the system into the error state, this is not enough to characterize the quality of the generated interfaces. Ideally, they should be validated by using some kind of oracle, such as descriptions of good and bad behaviors, usually defined by domain experts or encoded in a certain model.

For example, assuming the availability of the implementation of a partner service E_i, we could check whether $E_i \models \hat{I}_i$, where $\hat{I}_i$ is derived from $\hat{I}_{i_\pi}$, which is the interface specification computed for E_i. This check can be performed with a model checker, such as JavaPathFinder for Java-based implementations, or WS-Engineer for services implemented in BPEL. However, this approach may rarely be feasible in the realm of SBAs, since usually the implementations of the external services are not publicly available. Violations identified during such checks may signify either that a partner service is not appropriate for the desired composition, or that the interface generated may need to be refined. A domain expert would therefore need to inspect violations and decide on a course of action.

Domain expertise can also be used to validate directly the generated interfaces, to assess if they are either too strict or too weak, by analyzing the allowed (or disallowed) behaviors. In this sense, in section 6.4.1 we used our domain knowledge to (informally) claim that the interface generated for the *CPS* service was too restrictive.

Specific to the interface decomposition problem is to check if some behaviors, originally allowed by the global interface specification, are lost by the decomposition process. The lost behaviors can be discovered by checking the following relation: $Tr(\hat{I}) \subseteq Tr(||_{i=1}^{|E|} \hat{I}_i)$. This check can be performed with a model checker, such as LTSA. We expect this relation to not always hold, since some behaviors will be lost, as said above. However, when the check does not hold, the user can iteratively inspect each counterexample, to discriminate whether it represents a sink behavior, which cannot be realized in the actual system and thus can be ignored, or it is actually a missing behavior, which can then be added to the interface specification, which is thus refined.

6.5.2 Limitations of the Heuristic

In the *SCR* example, the interfaces we obtained for the partner services were satisfactory; however our experimentation has shown that this may not always be the case.

For example, consider an environment consisting of two services, E_1 and E_2, with E_1 providing operation c, and E_2 providing operations a and b. Assume the following LTS model represents the global interface:

```
S0 = (c -> S0 | b -> S1 | a -> S2),
S1 = (a -> S0 | b -> S1 | c -> S1),
S2 = (c -> ERROR | b -> S0 | a -> S2).
```

By decomposing this interface to compute $\hat{I}_1$ and $\hat{I}_2$, we notice that our heuristic blocks E_1 completely (no operation can be performed on it), while generates the universal interface for E_2.

More generally, our heuristic may block some good behaviors of the individual services, which instead could be safely allowed. This may happen because an operation of a service that directly leads to the error state, which is the one considered by our heuristic, may be actually triggered by an operation of another service. In the example above, the transition `c -> ERROR` is actually performed only after the transition `a -> S2` occurs; another heuristic could then allow E_1 to perform `c`, while the interface of E_2 could mandate the execution of `b` and `a` in this order.

6.6 Evaluation

The interface specifications decomposition technique has been implemented in the LTSA tool; here we report about the evaluation of our approach on two case studies. Each case study consisted of a service composition in the form of a BPEL process, of the syntactical interfaces (WSDL description) of the partner services of the composition, and of an informal description of the requirements that the composition had to fulfill.

The BPEL processes have been translated into the input format of the LTSA tool by means of WS-Engineer; the requirements have been first formalized in a temporal logic and then translated into an LTS description. The experiments have been executed on a computer running Apple Mac OS X 10.6.4 with a 2.16 GHz Intel Core 2 Duo processor and 2 GiB of memory.

6.6.1 Car Rental (full version)

This case study corresponds to the full-fledged version of the *SCR* example, with which it also shares the same requirements specification. The main difference lies in a fine-grained description of the BPEL process, which leads to more refined, and sometimes verbose, interface descriptions. For example, the two single transitions `lookupCar[0..1]` that in the running example correspond to invoking the `lookupCar` operation of the *CI* service and receiving, as output parameter, either 0 or 1, are expanded in a sequence of four operations: ⟨`cr_ci_invoke_lookupcar`, `cr_ci_receive_lookupcar`, `cr_ckr.condition.read.false`, `cr_ckr.condition.read.true`⟩.

If we consider this kind of expansion, we easily conclude that the interfaces generated are equivalent to, but bigger (in term of the size of the model) than the ones built obtained for the *SCR* example. For example, the interface of the *CI* service is the one showed in figure 6.4. The interface of the *CPS* service, as before, remains the universal interface.

In this example, the LTS model of the service composition contains 16 states and 20 transitions; the global interface specification contains 9 states and 22 transitions, and was built in 70 ms; the interface specifications of the services *CI* and *CPS* were built, respectively in 90 ms and 75 ms.

```
CIS = Q0,
Q0  = (cr_ci_invoke_markcarunavailable -> Q0
       |cr_ci_invoke_lookupcar -> Q2
       |cr_ci_invoke_markcaravailable -> Q3),
Q2  = (cr_ci_receive_lookupcar -> Q4),
Q3  = (cr_ci_invoke_markcarunavailable -> Q0
       |cr_ci_invoke_markcaravailable -> Q3
       |cr_ci_invoke_lookupcar -> Q5),
Q4  = (cr_ckr.condition.read.{false, true} -> Q0),
Q5  = (cr_ci_receive_lookupcar -> Q6),
Q6  = (cr_ckr.condition.read.false -> ERROR
       |cr_ckr.condition.read.true -> Q3).
```

Figure 6.4. **Interface computed for the *CI* service**

Validation against Original Specifications

The original example definition (see [32]) contained a set of property specifications of the behavior expected from the external services, manually written by the authors of the paper. We consider these properties as a possible oracle for evaluating how well our technique performs and thus we compared them with the ones generated by the tool.

The specification of the *CI*, called CIUpdate, service was:

> *"If the car is marked as available in the CI Service, and the car is not marked as unavailable until a* `lookupCar` *operation is invoked, then the* `lookupCar` *operation should return successfully".*

It is clear that this behavior is captured by the interface specification generated for the *CI* service.

For the *CPS* service, the specification was

> *"between two events signaling that the car exits from the parking lot, an event signaling the entrance for the same car must occur"*

It states the two events "car enter" and "car exit" should alternate. The interface specification obtained for this service, however, is the universal interface. In our opinion, this result is still correct, even if less useful, because the *CPS* service cannot be responsible for violations of the expected requirement.

In LTSA we have also implemented the possibility to search for and analyze lost behaviors, by checking $Tr(\hat{I}) \subseteq Tr(||_{i=1}^{|E|} \hat{I}_i)$. This check failed for the full *Car Rental* example, revealing one lost behavior whose trace is:

```
cr_ci_invoke_markcaravailable, cr_ci_invoke_lookupcar,
cr_ci_receive_lookupcar,cr_ckr.condition.read.false.
```

This trace can be interpreted as

> *"if the car is marked as available in the parking lot, then a request for the car will return a negative result"*,

which is an incorrect behavior. Note that this behavior is disallowed by the structure of the composite service, since `cr_ci_invoke_markcaravailable` will never be executed as the first action. Therefore we can safely state that this behavior has been added to the global interface specification through the completion with the sink state; it will never occur in the real system. This is the reason for which it is also missing from the interfaces derived for the partner services.

6.6.2 Order Booking

This case study has been taken from the set of processes distributed with the Oracle SOA Suite 10gR3. It consists of a process that is started when a customer places an order from a client web application. The process first inserts the order information in a database through the *ERPService*, then it retrieves customer information by invoking the *CustomerService*. The process checks the customer's credit card by invoking the *CreditService* and then determines if the order requires manual approval by invoking the *DecisionService (DS)*, which applies some business rules that take into account the status (platinum or not) of the customer. For orders that require manual approval, the process invokes the `requiresApproval` operation on the *Manager* Web service. When an order is approved, the process requests, in parallel, quotes from the suppliers, *SelectManufacturer* and *RapidService*, and then selects the supplier that responded with the lower quote. Afterwards, a shipping method is chosen by checking the amount of the order. After updating the order status on the database through the *ERPService*, the project sends a confirmation email to the customer, by invoking the *EmailService*, and then terminates.

A possible requirements specification for this composite service is:

> *"if a platinum customer places an order, it must be automatically approved; otherwise it must be approved manually"*.

This specification indirectly requires a certain behavior of the *DS* service, which we picked as the service for which to compute the interface specification.

We translated this specification into a property LTS and then applied the interface decomposition method based on the heuristic, to obtain the interface for the *DS* service. Although we omit its textual representation, in essence, it states that

> *"if a platinum customer places an order, then the return value will not be* manual approval*, and equivalently, if a non-platinum customer places an order, then the return value will not be* automatic-approval".

This specification matches the one informally described in the documentation of the example.

Since we were not interested in getting an individual interface specification for each of the other partner services, we generated an interface for them when considered as a whole and, as expected, we obtained the universal interface.

The LTS model of the composite service contains 80 states and 93 transitions; the global interface specification contains 29 states and 63 transitions, and was built in 76 ms; the interface specification of the *DS* service contains 6 states and 12 transitions, and was built in 82 ms. The interface for the rest of the components contains only one state, allowing all possible behaviors (i.e., it encodes the universal environment). A search for lost behaviors reveals two behaviors, which a manual inspection shows to be sink behaviors.

6.7 Summary

The correct behavior of a service composition, with respect to its requirements specification, depends on a certain, expected behavior of its partner services. However, most of the times the behavioral descriptions of the partner services are unknown. In this chapter, we presented our novel technique to automatically generating the behavioral interfaces of the partner services of a service composition, by decomposing the requirements specification of the composite services. We have formalized this problem, proposed a heuristic-based technique to solve it, implemented this technique in the LTSA tool, and applied it to two case studies.

Chapter 7

Incremental Verification: a Syntactic-Semantic Approach

7.1 Overview

The evolution of software systems is a well-known phenomenon in software engineering [100]. Software may evolve because of a change in the requirements or in the domain assumptions, leading to the development and deployment of many new versions of the software. This phenomenon is taken to extremes by open-world software, which is required to react to changes in its environment, by (self-) adapting its behavior while it is executing.

Support for these different kinds of software evolution should span through all the steps of the software development process; in this chapter we focus on the verification step. Incremental verification has been suggested as a possible approach to deal with evolving software [136]. An incremental verification approach tries to reuse as much as possible the results of a previous verification step, and accommodates within the verification procedure—possibly in a "smart" way—the changes occurring in the new version. By avoiding re-executing the verification process from scratch, incremental verification may considerably reduce the verification time. This may speed up change management, which may be subject to severe time constraints, especially if it needs to be performed at run time, to support dynamic self-adaptation.

In this chapter we present our proposal for incremental verification, the SiDECAR (Syntax-DrivEn inCrementAl veRification) framework. SiDECAR is a general framework to define verification procedures, which are automatically enhanced with incrementality by the framework itself. The framework follows a syntactic-semantic approach, since it assumes that the software artifact to be verified has a syntactic structure described by a formal grammar, and that the verification procedure is encoded as synthesis of semantic attributes [89], associated with the grammar and evaluated by traversing the syntax tree of the artifact. We based the framework on Floyd grammars (see [63] for the original definition as well as section 2.4 for a brief overview) because they allow for re-parsing, and hence semantic re-analysis, to be confined within an inner portion of the input that encloses the changed part.

This property is the key for an efficient incremental verification procedure: since the verification procedure is encoded within attributes, their evaluation proceeds incrementally, hand-in-hand with parsing.

The chapter is organized as follows. Section 7.2 shows how SiDECAR exploits Floyd grammars to support syntactic-semantic incremental verification. In section 7.3 we show SiDECAR at work, by encoding a standard verification procedure—reachability analysis—as semantic attributes of the grammar of a simple programming language; the verification procedure is then applied to two versions of an example program.

7.2 Syntactic-Semantic Incrementality

SiDECAR exploits a syntactic-semantic approach to define verification procedures that are encoded as semantic functions associated with an attribute grammar. In this section we show how Floyd grammars, equipped with a suitable attribute schema, can support incrementality in such verification procedures in a natural and efficient way.

The main reason for the choice of Floyd grammars is that, unlike other more modern and used grammars that support deterministic parsing, they enjoy the *locality property*, i.e., the possibility of starting the parsing from any arbitrary point of the sentence to be analyzed, independent of the context within which the sentence is located. It can be shown that, since the parsing of a Floyd grammar sentence is driven by precedence relations, a partial syntax tree corresponding to a derivation $\langle N\rangle \overset{*}{\Rightarrow} x$ can be deterministically built (in linear time) in a bottom-up way by using only a pair of single characters, say, $[\![a, b]\!]$ as the context of x (notice that necessarily a yields precedence to the first character of x and the last character of x takes precedence over b).

Consider a scenario where, after having built a syntax tree for a given input program (i.e., a certain input sentence), one or more parts of the programs are changed: thanks to the locality property only the changes should be re-parsed. Afterwards, the new local parse subtrees should be merged with the global parse tree using a suitable criterion. We say that the *matching condition* is satisfied when, after having parsed a substring, it is possible to identify the correct nesting point of its parse subtree within the global one.

The same locality property also supports parallel parsing, possibly to be exploited in a natural combination with incrementality: intuitively, the input can be split into many chunks that can be parsed in parallel by processes executing on different units. The results of the partial parsing processes can then be joined later on with great benefits in terms of performance [7].

As an intuitive example, consider the arithmetic expression ‘`5*4+2+6*7*8`’, derived from the grammar in figure 7.1 (equivalent to the one presented in section 2.4 and in-

$$\langle E\rangle ::= \langle E\rangle \text{ '}\mathbf{+}\text{' } \langle T\rangle \mid \langle T\rangle$$
$$\langle T\rangle ::= \langle T\rangle \text{ '}\ast\text{' '}\mathbf{n}\text{'} \mid \text{'}\mathbf{n}\text{'}$$

Figure 7.1. **Operator grammar for arithmetic expressions**

cluded here for convenience); the corresponding syntax tree is depicted in figure 7.2. Assume that the expression is modified in two points: the term `5*4` becomes `9`, while the term `6*7*8` becomes `7*8`. Their new parse subtrees can clearly be built independently, possibly in parallel; they are shown in figure 7.3a. The matching condition is also satisfied, since the merging points of the two subtrees can be identified as well, as depicted in figure 7.3b, where nesting points are emphasized in bold.

This nice property, which does not impose a strictly left-to-right parsing, has a price in terms of generative power. For example, the LR grammars traditionally used to describe and parse programming languages do not enjoy this property. However they can generate all the deterministic languages. Floyd grammars instead cannot. This limitation is more of theoretical interest than of real practical impact. Most programming languages in fact can be generated by a suitable Floyd grammar. For example, the original paper on Floyd grammars [63] details the minor adjustments required by the Algol 60 grammar to be treated as precedence grammar. More recently, to set up the benchmark adopted in [7] to evaluate the performances of a parallel parser for Floyd grammars, the original grammars of JSON, XML, and other languages needed only very minor changes to satisfy the Floyd definition.

In conclusion, Floyd grammars appear as a natural choice to support our syntactic-semantic approach to incremental (and possibly parallel) verification procedures.

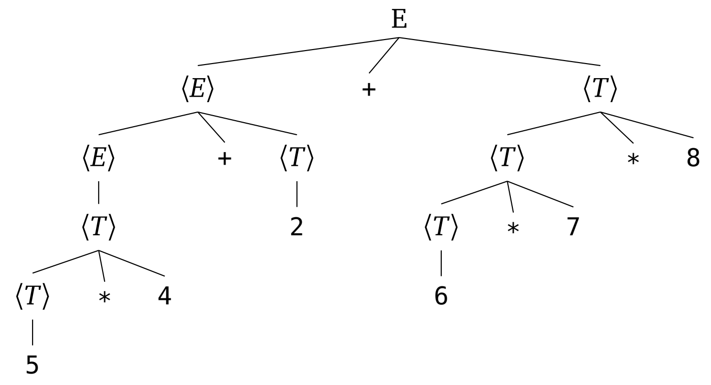


Figure 7.2. **Abstract syntax tree of the expression '`5*4+2+6*7*8`'**

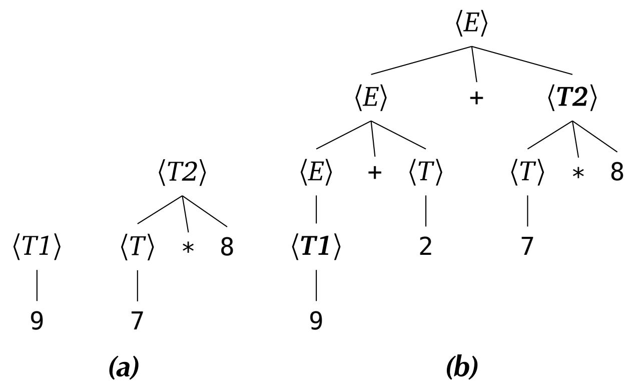


Figure 7.3. **Partial parse trees of the terms 7∗8 and 9 and their nesting within the global syntax tree**

7.2.1 Syntactic Incrementality

The intuition behind the example in figure 7.3 can be generalized and formalized to obtain a general procedure for incremental parsing of Floyd grammars.

Consider a generic syntax tree as the one depicted in figure 7.4, in which the non-terminal $\langle N\rangle$ generates the string xwz. Suppose that substring w is replaced by a new string w'. An algorithm that considers the operator precedence relations of a Floyd grammar can restart parsing from the substring w' and its context $[\![x, z]\!]$, regardless of the rest of the input. Reductions are applied by finding the innermost pair of $\lessdot \cdots \gtrdot$ that overlaps with w', possibly with $\doteq$ relations in between, and then proceeding both rightward and leftward until a matching condition is satisfied, as described in [61]. Suppose that the parsing of $xw'z$ leads to the derivation $\langle N\rangle \overset{*}{\Rightarrow} xw'z$, with the same non-terminal $\langle N\rangle$ as in $\langle N\rangle \overset{*}{\Rightarrow} xwz$: we say that the matching condition is satisfied. Since the remaining part of the tree is not affected by the change in the input string, the parsing process is completed after the old subtree rooted in $\langle N\rangle$ is replaced with the new one.

A similar procedure can be applied in case of multiple changes to the input string, with the possibility of supporting also parallel parsing. If the subtrees affected by the changes are disjoint, i.e., their contexts do not overlap as depicted in figure 7.5, the tasks of parsing the two new substrings $xw'z$ and $yv's$ can be performed by two processes and completed in a totally independent way. In case the two subtrees share at least one node, the matching condition is not satisfied, and the two partial parsings have to be merged together. The two changes can be re-parsed separately until the respective subtrees share at least one node. Once the two processes executing the parsing are about to apply a reduction involving (at least) one node shared by both subtrees, the control is passed to only one of the processes, which then completes the

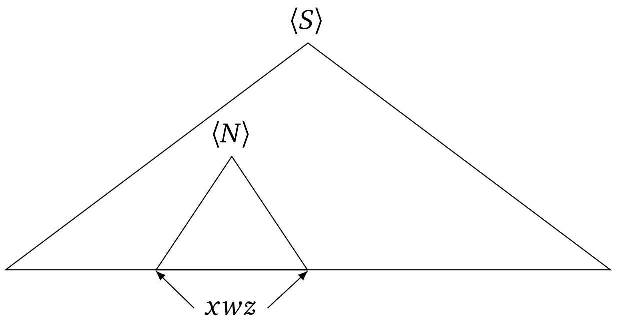


Figure 7.4. **Syntax tree rooted in the axiom ⟨S⟩, with a subtree rooted in the non-terminal ⟨N⟩ generating the string *xwz***

parsing by itself. This simple strategy can be significantly enhanced by means of a more precise identification of the nodes that effectively need to be changed, e.g., by jointly applying $LR \cap RL$ techniques [70].

In the current prototypal implementation of SiDECAR, the incremental parser for Floyd grammars has the following complexities: $O(n)$, with n being the length of the string, in case of parsing from scratch; $O(m)$, with m being the size of the *modified subtree(s)*, in case of incremental parsing; $O(1)$ for the matching condition test.

7.2.2 Semantic Incrementality

In a bottom-up parser, semantic actions are performed during a reduction. This allows the re-computation of semantic attributes after a change to proceed hand-in-hand with the re-parsing of the modified substring. Suppose that, after replacing w with w', incremental re-parsing builds a derivation $\langle N \rangle \overset{*}{\Rightarrow} xw'z$, with the same non-terminal $\langle N \rangle$ as in $\langle N \rangle \overset{*}{\Rightarrow} xwz$, so that the matching condition is verified. Assume also that $\langle N \rangle$ has an attribute α_N. Two situations may occur related to the computation of α_N:

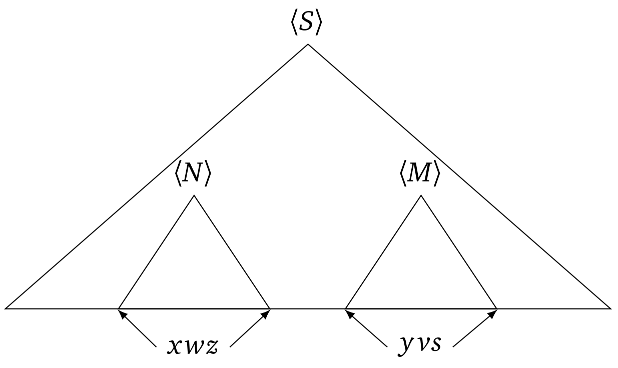


Figure 7.5. **Parallel incremental update of a parse tree**

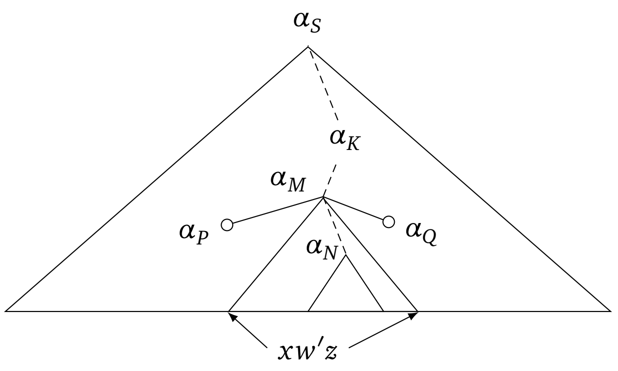


Figure 7.6. **Incremental evaluation of semantic attributes**

1. The α_N attribute associated with the new subtree rooted in $\langle N \rangle$ has the same value as before the change. In this case, all the remaining attributes in the rest of the tree will not be affected, and no further analysis is needed.

2. Despite the syntactic matching, the new value of α_N is different from the one it had before the change. In this case, as suggested by figure 7.6, only the attributes on the path from $\langle N \rangle$ to the root $\langle S \rangle$ (e.g., $\alpha_M, \alpha_K, \alpha_S$) can change and thus need to be recomputed. The values of the other attributes not on the path from $\langle N \rangle$ to the root (e.g., α_P and α_Q) do not change: there is no need to recompute them.

7.3 SiDECAR at Work

In this section we show how to use SiDECAR by defining a verification procedure for reachability analysis in control flows.

The first step to use SiDECAR is to define a Floyd grammar, from which the artifacts (e.g., the programs) to be analyzed can be generated; we use programs written in the *Mini* language, whose grammar is shown in figure 7.7. The *Mini* language includes the major constructs of structured programming [41], from which one can derive modern imperative programming languages as well as languages for defining workflows of service compositions. For the sake of readability and to reduce the complexity of the attribute schema, *Mini* programs support only (global) boolean variables and boolean functions (with no input parameters). These assumptions can be relaxed, with no impact on the applicability of the approach.

To show the benefits of incrementality, we detail the execution of reachability analysis on two versions of the same example program. The two versions of the example program are shown in figure 7.8; they differ in the assignment at line 3, which determines the execution of the subsequent *if* statement, with implications on the results of the analysis. Figure 7.9 depicts the syntax tree of version 1 of the program, as well

⟨*S*⟩ ::= ‘**`begin`**’ ⟨*stmtlist*⟩ ‘**`end`**’

⟨*stmtlist*⟩ ::= ⟨*stmt*⟩ ‘**`;`**’ ⟨*stmtlist*⟩
 | ⟨*stmt*⟩ ‘**`;`**’

⟨*stmt*⟩ ::= ⟨*function-id*⟩ ‘**`(`**’ ‘**`)`**’
 | ⟨*var-id*⟩ ‘**`:=`**’ ‘**`true`**’
 | ⟨*var-id*⟩ ‘**`:=`**’ ‘**`false`**’
 | ⟨*var-id*⟩ ‘**`:=`**’ ⟨*function-id*⟩ ‘**`(`**’ ‘**`)`**’
 | ‘**`if`**’ ⟨*cond*⟩ ‘**`then`**’ ⟨*stmtlist*⟩ ‘**`else`**’ ⟨*stmtlist*⟩ ‘**`endif`**’
 | ‘**`while`**’ ⟨*cond*⟩ ‘**`do`**’ ⟨*stmtlist*⟩ ‘**`endwhile`**’

⟨*var-id*⟩ ::= ...

⟨*function-id*⟩ ::= ...

⟨*cond*⟩ ::= ...

Figure 7.7. **The grammar of the *Mini* language**

as the subtree that is different in version 2; nodes of the tree have been numbered for quick reference.

The second step in using SiDECAR is to define the attribute schema that encodes the analysis to be performed, in terms of the algorithm and of the data structures. Although these items are specific to each analysis, the framework is general enough to provide a common infrastructure that supports syntactic-semantic incrementality for any analysis defined on the top of it.

Before describing reachability analysis and the corresponding attribute schema, here we introduce some useful notations. Given a *Mini* program P, F_P is the set of boolean functions and V_P the set of boolean variables defined within P; E_P is the set of boolean expressions that can appear as the condition of an *if* or a *while* statement in P. An expression $e \in E_P$ is either a combination of boolean predicates on program variables or a placeholder predicate labeled $*$. Hereafter, we drop the subscript P in F_P, V_P, and E_P whenever the program is clear from the context.

7.3.1 Reachability Analysis

Reachability analysis is a basic software model checking procedure, which solves the safety verification problem: given a program and a safety property, we want to decide whether there is an execution of the program that leads to a violation of the property.

In software model checking, it is common to use a transition-relation representation of programs [79], in which a program is characterized by a set of (typed) variables, a set of control locations (including an initial one), and a set of transitions, from a control location to another one, labeled with constraints on variables and/or with program operations. Examples of this kind of representation are control-flow graphs [1]

```
begin
 opA();
 x := true;
 if (x==true)
  then opB();
  else opA();
 endif;
end
```

(a) **Version 1**

```
begin
 opA();
 x := false;
 if (x==true)
  then opB();
  else opA();
 endif;
end
```

(b) **Version 2**

Figure 7.8. **The two versions of the example program**

and control-flow automata [18]. A state of the program is characterized by a location and by the valuation of the variables at that location. A computation of the program is a (finite or infinite) sequence of states, where the sequence is induced by the transition relation over locations. Checking for a safety property can be reduced to the problem of checking for the reachability of a particular location, the *error* location, for example, by properly instrumenting the program code according to the safety specification.

In the implementation of reachability analysis with SiDECAR we assume that the safety property is defined as a *property automaton* [46], whose transitions correspond either to a procedure call or to a function call that assigns a value to a variable. From this automaton we then derive the corresponding *image automaton*, which traps violation of the property in an error location (called *ERR*).

Formally, let *VA* be the set of variable assignments from functions, i.e., $VA = \{x := f \mid x \in V \text{ and } f \in F\}$. A property automaton A is a quadruple $A = \langle S, T, \delta, s_0 \rangle$ where S is a set of locations, T is the alphabet $T = F \cup VA$, δ is the transition function $\delta : S \times T \to S$, and s_0 is the initial location. Given a property automaton A, the corresponding image automaton A' is defined as $A' = \langle S \cup \{ERR\}, T, \delta', s_0 \rangle$, where $\delta' = \delta \cup \{(s, t, ERR) \mid (s, t) \in S \times T \land \neg\exists s' \in S \mid (s, t, s') \in \delta\}$. An example of a property automaton specifying the alternation of operations `opA` and `opB` on sequences starting with `opA` is depicted in figure 7.10; transitions drawn with a dashed line are added to the property automaton to obtain its image automaton.

Instead of analyzing the program code instrumented with the safety specification, we check for the reachability of the error location in an execution trace of the image automaton, as induced by the syntactic structure of the program.

More specifically, each location of the automaton is paired with a configuration of the program, which consists of a mapping of the program variables and of the traversal conditions for the paths taken so far. A configuration is invalid if the set of predicate conditions holding at a certain location of the program is not compatible with the

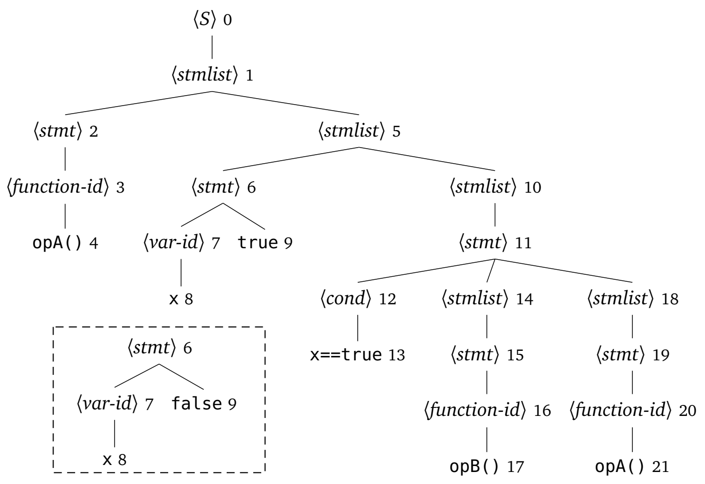


Figure 7.9. **The syntax tree of version 1 of the example program; the subtree in the dashed box shows the difference (node 9) in the syntax tree of version 2**

current variables mapping for that location. Formally, let $VM : V \mapsto \{true, false\}$ be a mapping from program variables to their value (if defined). The set of possibile configurations that can be reached during the execution of a program is denoted by $C = (VM \times E) \cup \{\bot\}$, where $\bot$ stands for an invalid configuration.

Configurations of the program may change when variables are assigned a new value, e.g., by a direct assignment of a literal or by assigning the return value of a function. We use a function *upd* that updates a configuration and checks whether it is valid or not. The function *upd* is defined as $upd\colon (C \times V \cup \{\varepsilon\} \times \{true, false\} \cup \{\varepsilon\} \times E \cup \{\varepsilon\} \times \{true, false\} \cup \{\varepsilon\}) \rightarrow C$. The function takes a configuration, a variable, its new value, a combination of boolean expressions (corresponding to a certain path condition), its new value, and returns the new configuration; the ε symbol accounts for empty parameters.

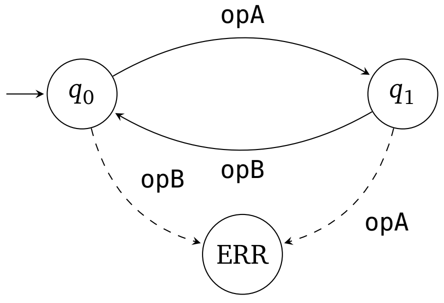


Figure 7.10. **A property automaton; dashed lines belong to the corresponding image automaton**

We call the pair ⟨location of the image automaton, configuration of the program⟩ an *extended state*. A safety property represented as an image automaton is violated if it is possible to reach from the initial extended state another extended state whose location component is the *ERR* location. Each statement in the program defines a transition from one extended state to another.

For example, a procedure call determines the location component in an extended state by following the transition function of the image automaton corresponding to the call. An assignment to a variable updates the program configuration component of an extended state. In case a variable is assigned the return value of a function invocation, both components of an extended state are updated.

Conditions in selection and loop statements are evaluated and the program configuration of the corresponding extended state is updated accordingly, to keep track of which path conditions have been taken. For an *if* statement, we keep track of which extended states could be reachable by executing the statement, considering both the *then* branch and the *else* branch. For a *while* statement, we make the common assumption that a certain constant K is provided to indicate the number of unrolling passes of the loop. We then keep track of which extended states could be reachable, both in case the loop is not executed and in case the loop is executed K times.

7.3.2 Attribute Schema

The set of attributes is defined as:

- $SYN(\langle S\rangle) = SYN(\langle stmlist\rangle) = SYN(\langle stmt\rangle) = \{\gamma\}$;

- $SYN(\langle cond\rangle) = \{\gamma, v\}$;

- $SYN(\langle var\text{-}id\rangle) = SYN(\langle function\text{-}id\rangle) = \{\eta\}$;

where:

- $\gamma \subseteq S \times C \times S \times C$ is the relation that defines a transition from one extended state to another one;
- v is a string corresponding to the literal value of an expression $e \in E$;
- η is a string corresponding to the literal value of an identifier.

For the γ attribute of non-terminal ⟨*cond*⟩ we use the symbol γ^T (respectively γ^F) to denote the attribute γ evaluated when the condition ⟨*cond*⟩ is *true* (respectively, *false*). We also define the operation of composing γ relations (denoted by the $\circ$ operator) as follows: $\gamma_1 \circ \gamma_2 = \langle s_1, c_1, s_2, c_2\rangle$ such that there exist $\langle s_1, c_1, s_i, c_i\rangle \in \gamma_1$ and $\langle s_i, c_i, s_2, c_2\rangle \in \gamma_2$. The attribute schema is defined as follows, where we use the symbols s, s_1, s_2 and c, c_1, c_2 to denote generic elements in S and C, respectively.

1. $\langle S\rangle$ ::= '**begin**' $\langle stmtlist\rangle$ '**end**'
   $\gamma(\langle S\rangle) := \gamma(\langle stmtlist\rangle)$

2. (a) $\langle stmtlist_0\rangle$ ::= $\langle stmt\rangle$ '**;**' $\langle stmtlist_1\rangle$
   $\gamma(\langle stmtlist_0\rangle) := \gamma(\langle stmt\rangle) \circ \gamma(\langle stmtlist_1\rangle)$
   (b) $\langle stmtlist\rangle$ ::= $\langle stmt\rangle$ '**;**'
   $\gamma(\langle stmtlist\rangle) := \gamma(\langle stmt\rangle)$

3. (a) $\langle stmt\rangle$ ::= $\langle function\text{-}id\rangle$ '**(**' '**)**'
   $\gamma(\langle stmt\rangle) := \langle s_1, c, s_2, c\rangle$ such that there is $f \in F$ with $\delta(s_1, f) = s_2$ and $\eta(\langle function\text{-}id\rangle) = f$
   (b) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '**:=**' '**true**'
   $\gamma(\langle stmt\rangle) := \langle s, c_1, s, c_2\rangle$ with $c_2 = upd(c_1, \eta(\langle var\text{-}id\rangle), true, \varepsilon, \varepsilon)$
   (c) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '**:=**' '**false**'
   $\gamma(\langle stmt\rangle) := \langle s, c_1, s, c_2\rangle$ with $c_2 = upd(c_1, \eta(\langle var\text{-}id\rangle), false, \varepsilon, \varepsilon)$
   (d) $\langle stmt\rangle$ ::= $\langle var\text{-}id\rangle$ '**=**' $\langle function\text{-}id\rangle$ '**(**' '**)**'
   $\gamma(\langle stmt\rangle) := \langle s_1, c_1, s_2, c_2\rangle \cup \langle s_1, c_1, s_2, c_3\rangle$ such that there is $f \in F$ with $\delta(s_1, f) = s_2$, $\eta(\langle function\text{-}id\rangle) = f$, $c_2 = upd(c_1, \eta(\langle var\text{-}id\rangle), true, \varepsilon, \varepsilon)$, and $c_3 = upd(c_1, \eta(\langle var\text{-}id\rangle), false, \varepsilon, \varepsilon)$
   (e) $\langle stmt\rangle$ ::= '**if**' $\langle cond\rangle$ '**then**' $\langle stmlist_0\rangle$ '**else**' $\langle stmlist_1\rangle$ '**endif**'
   $\gamma(\langle stmt\rangle) := \gamma^T(\langle cond\rangle) \circ \gamma(\langle stmtlist_0\rangle) \cup \gamma^F(\langle cond\rangle) \circ \gamma(\langle stmtlist_1\rangle)$
   (f) $\langle stmt\rangle$ ::= '**while**' $\langle cond\rangle$ '**do**' $\langle stmtlist\rangle$ '**endwhile**'
   $\gamma(\langle stmt\rangle) := \gamma^{body} \circ \gamma^F(\langle cond\rangle)$ where $\gamma^{body} = \left(\gamma^T(\langle cond\rangle) \circ \gamma(\langle stmtlist\rangle)\right)^K$

4. $\langle cond\rangle$ ::= ...
   $\gamma(\langle cond\rangle) := \gamma^T(\langle cond\rangle) \cup \gamma^F(\langle cond\rangle) = \langle s, c_1, s, c_2\rangle \cup \langle s, c_1, s, c_3\rangle$ where $c_2 = upd(c_1, \varepsilon, \varepsilon, \nu(\langle cond\rangle), true)$ and $c_3 = upd(c_1, \varepsilon, \varepsilon, \nu(\langle cond\rangle), false)$

7.3.3 Application to the Example

We show how to perform reachability analysis with SiDECAR on the two versions of the example program. For both examples, we consider the safety property specified with the automaton in figure 7.10. In the steps of attribute synthesis, for brevity, we use numbers to refer to the corresponding nodes in the syntax tree.

Example Program - Version 1

Given the abstract syntax tree depicted in figure 7.9, attributes are synthesized as follows:

- $\gamma(2) := \{\langle q_0, c, q_1, c\rangle, \langle q_1, c, ERR, c\rangle\}$

- $\gamma(6) := \langle s, c_1, s, upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon)\rangle$
- $\gamma(12) := \gamma^T(12) \cup \gamma^F(12) :=$
  $\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle \cup \langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false)\rangle$
- $\gamma(15) := \{\langle q_1, c, q_0, c\rangle, \langle q_0, c, ERR, c\rangle\}$
- $\gamma(14) := \gamma(15)$
- $\gamma(19) := \{\langle q_0, c, q_1, c\rangle, \langle q_1, c, ERR, c\rangle\}$
- $\gamma(18) := \gamma(19)$
- $\gamma(11) := \gamma^T(12) \circ \gamma(14) \cup \gamma^F(12) \circ \gamma(18) :=$
  $\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle \circ \{\langle q_1, c, q_0, c\rangle, \langle q_0, c, ERR, c\rangle\}$
  $\cup$
  $\langle s, c_1, s, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false)\rangle \circ \{\langle q_0, c, q_1, c\rangle, \langle q_1, c, ERR, c\rangle\} :=$
  $\{\langle q_1, c_1, q_0, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle,$
  $\langle q_0, c_1, ERR, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle,$
  $\langle q_0, c_1, q_1, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false)\rangle,$
  $\langle q_1, c_1, ERR, upd(c_1, \varepsilon, \varepsilon, \texttt{"x==true"}, false)\rangle\}$
- $\gamma(10) := \gamma(11)$
- $\gamma(5) := \gamma(6) \circ \gamma(10) :=$
  $\{\langle q_1, c_1, q_0, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle,$
  $\langle q_0, c_1, ERR, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle,$
  $\langle q_0, c_1, q_1, \perp\rangle,$
  $\langle q_1, c_1, ERR, \perp\rangle\}$

The last two tuples of $\gamma(5)$ are discarded because they contain a $\perp$ configuration. The $\perp$ component of this configuration is returned by *upd*; according to its semantics, the evaluation of the condition `"x==true"` to *false* is not compatible with the previous configuration, where `x` is assigned the value *true*. Hence, we have:

- $\gamma(5) := \{\langle q_1, c_1, q_0, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle,$
  $\langle q_0, c_1, ERR, upd(upd(c_1, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle\}$
- $\gamma(1) := \gamma(2) \circ \gamma(5) := \langle q_0, c, q_0, upd(upd(c, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle$
- $\gamma(0) = \gamma(1) = \langle q_0, c, q_0, upd(upd(c, \texttt{"x"}, true, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, true)\rangle$

The resulting $\gamma(0)$ shows that the error location is not reachable from the initial extended state. Therefore we can conclude that the property will not be violated by any execution of the program.

Example Program - Version 2

Version 2 of the example program differs from version 1 only in the assignment at line 3, reflected in node 9 of the subtree shown in the box of figure 7.9. This change in the syntax tree triggers the restart of parsing from the node, leading to the derivation of the non-terminal $\langle stmt \rangle$ at node 6, which satisfies the matching condition. The corresponding re-computation of the attributes proceeds from node 6 up to the root requiring only the following steps:

- $\gamma(6) := \langle s, c_1, s, upd(c_1, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon)\rangle$
- $\gamma(5) := \gamma(6) \circ \gamma(10) :=$
  $\{\langle q_1, c_1, q_0, \perp\rangle,$
  $\langle q_0, c_1, \mathit{ERR}, \perp\rangle,$
  $\langle q_0, c_1, q_1, upd(upd(c_1, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle$
  $\langle q_1, c_1, \mathit{ERR}, upd(upd(c_1, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle\ \}$

The first two tuples of $\gamma(5)$ are discarded because they contain a $\perp$ configuration. The $\perp$ component of this configuration is returned by *upd*; according to its semantics, the evaluation of the condition `"x==true"` to *true* is not compatible with the previous configuration, where `x` is assigned the value *false*. Hence, we have:

- $\gamma(5) := \{\langle q_0, c_1, q_1, upd(upd(c_1, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle,$
  $\langle q_1, c_1, \mathit{ERR}, upd(upd(c_1, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle\}$
- $\gamma(1) := \gamma(2) \circ \gamma(5) :=$
  $\langle q_0, c, \mathit{ERR}, upd(upd(c, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle$
- $\gamma(0) = \gamma(1) = \langle q_0, c, \mathit{ERR}, upd(upd(c, \texttt{"x"}, \mathit{false}, \varepsilon, \varepsilon), \varepsilon, \varepsilon, \texttt{"x==true"}, \mathit{false})\rangle$

Note that we reuse results from the analysis of version 1, since $\gamma(10)$ and $\gamma(2)$ have not changed. Although the example and its state space are small and not representative, in the analysis of version 2 we processed only 7 tuples of the state space, compared with the 26 ones processed for version 1: a reduction of about 75% of the state space.

Finally, looking at $\gamma(0)$ we notice that the error location is actually reachable, which means that version 2 of the program violates the safety property.

Despite its small size, the example gives a glimpse of the benefits of incrementality in the SiDECAR approach, showing how the evaluation of each change in the input program triggers only the recomputation of the semantic attributes affected by the change, allowing for a high reuse of the previous results.

7.4 Summary

In this chapter we presented SiDECAR, our framework supporting a syntactic-semantic approach for incremental verification. We also showed its application in the definition of an incremental procedure for reachability analysis.

SiDECAR has only two usage requirements: 1) the artifact to be verified should have a syntactic structure derivable from a Floyd grammar; 2) the verification procedure has to be formalized as synthesis of semantic attributes. The expressiveness of Floyd grammars and the well-known versatility of attribute grammars guarantee that there is no practical limitation in using SiDECAR. Moreover, incrementality is automatically provided by the framework without any further effort for the developer.

The parsing algorithm used within SiDECAR has a temporal complexity linear in the length of the changes to be analyzed. Hence any change in the program has a minimal impact on the adaptation of the abstract syntax tree, without any further constraint on the grammar. Semantic incrementality allows for low-impact (re)evaluation of the attributes, by proceeding along the path from the node corresponding to the change to the root, whose length is normally logarithmic with respect to the length of the program. The use of SiDECAR may result in a significant reduction of the reanalysis and semantic re-evaluation steps. The saving can be very relevant in the case of large programs and rich and complex attributes schema.

We remark that the example presented in section 7.3 was not designed to be directly applied to real-world analysis, but to show, in a simple and readable way, the generality of the approach. However, the generality and flexibility of Floyd grammars allow for using in a natural way much richer languages than the *Mini* example used here; on the other hand, having attribute grammars the same computational power as Turing machines, they enable formalizing in this framework any algorithmic schema at any sophistication and complexity level. For example, in the case of reachability analysis, a more elaborated attribute schema could support both new language features (e.g., heap data structures) and different verification algorithms (e.g., abstraction-based techniques).

Chapter 8

Intermezzo 2: Verification - State of the Art

In this chapter we report on the state of the art related to interface decomposition (section 8.1) and to incremental verification (section 8.2).

8.1 On Interface Decomposition

The work presented in chapter 6 is closely related to the problem of synthesizing individual service behaviors from a choreography specification, such as conversation protocols [66], WS-CDL models [127], and collaboration diagrams [131]. These approaches define a *projection* operation that derives the implementations of the participating peers by filtering the global specification on the actions alphabet of each peer, which is similar in spirit to the basic decomposition approach described in Section 6.4.1; additionally, in [131], extra communication actions among the generated peers are added in case some behaviors may not be realizable in a distributed fashion. The difference with our work lies in the point of view adopted: the aforementioned approaches consider a superset of the possible behaviors and narrow it down to achieve the exact behavior dictated by the choreography specification. In our work, we view the global interface as the maximum behavior that could be allowed for the composition based on a property, and we generate a subset of the possible behaviors. Our process is driven by the error behaviors that have to be blocked; error traces guide us in the heuristic to assign to partner services the responsibility of blocking those behaviors. Still related to the synthesis problem, reference [104] shows, in the context of verification of choreographies expressed in BPEL4CHOR, how a single participant of a choreography can be synthesized starting from the description of the choreography and from the BPEL models of the other participants. Besides the limitation of synthesizing at most one participant, this work makes the assumption that the BPEL models of the other participants are available; this assumption is unrealistic in the context of open-world services. In defining our interface decomposition technique, we have assumed synchronous interactions among services. For asynchronous systems, recently Basu et al. [15] have proved the decidability of the choreography realizability problem for systems communicating through asynchronous messages and unbounded FIFO message queues. The problem of synthesizing distributed transition systems from global specifications has been dealt with in [139]. This work assume the knowledge of the complete distribution structure of the system; however, this is not a reasonable working assumption in the context of SBAs, where partner services are usually seen as black-boxes.

The problem of generating the interface of the environment of a system, given a property it should satisfy, has been originally dealt with in [72], in the context of model checking. However, the approach generates only the global interface, not the interfaces of the individual components of the system. Other work [44] describes a compositional reasoning approach for the verification of middleware-based software architecture descriptions. Given a graphical scenario of the architecture of a generic application in terms of Message Sequence Charts (MSCs), the approach tries to verify the global property by verifying local properties of the architectural components. This last step requires to decompose the global property into local properties; the decomposition is based on the analysis of the structure of the MSCs, which is similar to our heuristic that considers the structure of the global interface specification.

The use of a description of the system requirements to generate behavioral models of the system components is also common in the context of behavioral model synthesis. One approach [52] proposes to inductively synthesize the LTS models of each system component from a set of end-users scenarios, both positive and negative, in the form of MSCs. The approach operates at the stage of requirements, where users can interactively refine the scenario-based description by answering questions; in our work, we assume the requirements are fixed and thus rely on the accuracy of the specification to get expressive interfaces. The approach presented in [4] derives operational requirements (in the form of pre- and trigger-conditions) from goal models, using a combination of model checking, inductive learning and manual elaboration of scenarios; however, the approach does not support learning the operational requirements for an individual component of a system. In [94], behavioral models, in the form of Modal Transition Systems, are generated at the component level from a set of scenarios and property specifications. The algorithm assumes that domain variables are used for defining the pre- and post-conditions of component operations; however, for service components, pre- and post-conditions might not available. Another technique [142] constructs behavioral models (in the form of Modal Transition Systems) from both safety properties and scenario-based specifications; however, the models generated are at the system level, not at the component level.

Other approaches perform decomposition of a global specification either to reduce the size of the model to verify by means of slicing [95, 43], or to support compositional verification for systems that are not structured into parallel components [112].

Inferring the specifications of components is also a goal shared with program specification miners, such as Adabu [51] and GK-tail [105]. These approaches usually perform static analysis, code instrumentation and analysis of the execution traces to derive the usage patterns of components and thus need to access the code of the components for which you want to discover the specification. This latter step is not feasible in the domain of service-oriented computing. In the context of Web services, the Strawberry approach [17] derives the behavioral model of a service by analyzing its syntactical interface and applying a combination of graph synthesis, heuristics and testing. However, all these approaches consider the behavior of a single service (component) in isolation, while we are interested in discovering the behavioral interfaces of components that guarantee the requirements of the composite application.

8.2 On Incremental Verification

Different methodologies have been proposed in the literature as the basis for incremental[1] verification techniques. They are mainly grounded in the assume-guarantee [80] paradigm. This paradigm views systems as a collection of cooperating modules, each of which has to guarantee certain properties. The verification methods based on this paradigm are said to be compositional, since they allow reasoning about each module separately and deducing properties about their integration. If the effect of a change can be localized inside the boundary of a module, the other modules are not affected, and their verification does not need to be redone. This feature is for example exploited in [47], which proposes a framework for performing assume-guarantee reasoning in an incremental and fully automatic fashion.

A second approach to incrementality is based on anticipating the changes that may occur in the system. It does not rely on a precise modular structure of the system nor suffers for the percolation of changes' effects through interfaces; it is based on the notion of *partial evaluation*, originally introduced in [58]. Partial evaluation can be seen as a transformation from the original version of the program to a new version called *residual program*, where the properties of interest have been partially computed against the static parts, preserving the dependency on the variable ones. As soon as a change is observed, the computation can be moved a further step toward completion by fixing one or more variable parts according to the observations.

Other approaches for incremental verification based on (regression) model checking reason in terms of the representation (e.g., a state-transition system) explored during the verification, by assessing how it is affected by changes in the program. The main idea is to maximize the reuse of the state space already explored for previous versions of the program, isolating the parts of the state space that have changed in the

[1]Incidentally, the use of the term *incremental model checking* in the context of bounded model checking [38] has a different meaning, since it refers to the possibility of changing the bound of the checking.

new version. The first work in this line of research addressed modal mu-calculus [138]. Henzinger et al. [78] analyze a new version of the program by checking for the conformance of its (abstract) state space representation with respect to the one of the previous version. When a discrepancy is found, the algorithm that recomputes the abstraction is restarted from that location. Depending on where the change is localized in the program text, the algorithm could invalidate—and thus recompute—a possibly large portion of the program state space.

Incremental approaches for explicit-state model checking of object-oriented programs, such as [98] and [146], analyze the state space checked for a previous version and assess, respectively, either the transitions that do not need to be re-executed in a certain exploration of the state space, or the states that can be pruned, because not affected by the code change. These approaches tie incrementality to the low-level details of the verification procedure, while SiDECAR supports incrementality at a higher level, independently from the algorithm and data structures defined in the attributes. Conway et al. [48] define incremental algorithms for automaton-based safety program analyses. Their granularity for the identification of reusable parts of the state space is coarse-grained, since they take a function as the unit of change, while SiDECAR has a finer granularity, at the statement level. A combination of a modular verification technique that also reuse cached information from the checks of previous versions is presented in [93] for aspect-oriented software.

The syntactic-semantic approach embedded in SiDECAR does not constrain incrementality depending on on the modular structure of the artifacts, as instead required by assume-guarantee approaches. Furthermore, it provides a general and unifying methodology for defining verification procedures for functional and non-functional requirements.

Part IV

Reputation Management

Chapter 9

Reputation Management of Composite Services

9.1 Overview

The dependability of composite SBAs is largely affected by their constituent services. Composite services have to adapt to the open, dynamically changing environment where remote services may fail or new services may be offered at any moment. The ability to bind to required services at run time is a key mechanism to cope with the challenges of open-world software. As the market of available services for a given functionality changes over the time, composite services that depend on that functionality need to evolve, adapting their service bindings so as to leverage the best performing services currently available.

Selecting the best service for a required functionality presupposes a reliable and efficient mechanism to provide the service rankings. For this purpose, *reputation mechanisms* have been proposed [114]. They collect clients' ratings on experienced service behavior to compute the actual QoS delivered to clients and to rank functionally-equivalent services accordingly. Reputation mechanisms therefore promote the sharing of service monitoring information amongst clients. Researchers have shown that reputation mechanisms can be designed to provide incentives that make honest reporting rational for the clients [84].

However, current standard environments for the execution of composite services, such as BPEL engines, do not integrate any reputation mechanisms. Although it is possible to program composite services that explicitly interact with a reputation mechanism so as to report feedback on service interactions and to dynamically choose the most efficient services, the needed development effort is prohibitive in practice. Moreover, feedback reporting on both experienced service functionality and QoS presumes an appropriate monitoring infrastructure.

To overcome these issues, we designed ReMan, a reputation management infrastructure that serves the following goals:

1. To provide a mechanism for assessing service behavior and ranking functionally-equivalent services based on past interactions with these services by other clients.

2. To support user notifications when particular reputation-related events occur, allowing for an early discovery of possible failure situations.

3. To ensure reputation-enabled execution of composite services in a way that is completely transparent to the programmer, who can concentrate exclusively on the functional aspects of the composite service.

4. To allow for an open and extensible platform supporting a high degree of customization of the way services' reputation is computed.

The architecture of ReMan enables the transparent integration of reputation mechanisms in standard execution environments for composite services. It includes a customizable reputation mechanism that is integrated with a customizable UDDI [118] service repository, thus enabling reputation-aware service selection. The execution environment running the composite service (a BPEL engine) is instrumented for monitoring service invocations and reporting feedback to a reputation mechanism that computes services' reputation.

ReMan supports subscriptions for service functionalities, resulting in notifications upon changes in service reputation and upon the availability of better performing services for a given functionality, respectively. These notifications enable the automated update of service bindings, ensuring the automated evolution of composite service in response to a dynamically changing service market. Although, for example, upon receiving the notification that the reputation of a service has dropped below a given threshold, a client could replace the affected service and hence avoid possible problems before they actually occur, in this chapter we focus on the generic reputation infrastructure itself and do not address the concrete actions taken upon reputation-related events, since these actions are specific to client policies.

To validate and evaluate our approach, we measured the overhead generated by ReMan both for deployment and execution of composite services. We explore separately the different aspects of our instrumentation, service execution monitoring and communication with an external reputation mechanism, to assess how much each of them contributes to the overall observed overhead.

The rest of the chapter is organized as follows. Section 9.2 describes the architecture of ReMan. Section 9.3 presents the technique we use to estimate service reputation. Section 9.4 explains the implementation of the main components. Section 9.5 presents the results of our experimental evaluation. Section 9.6 concludes part IV by surveying related work in the area of reputation management.

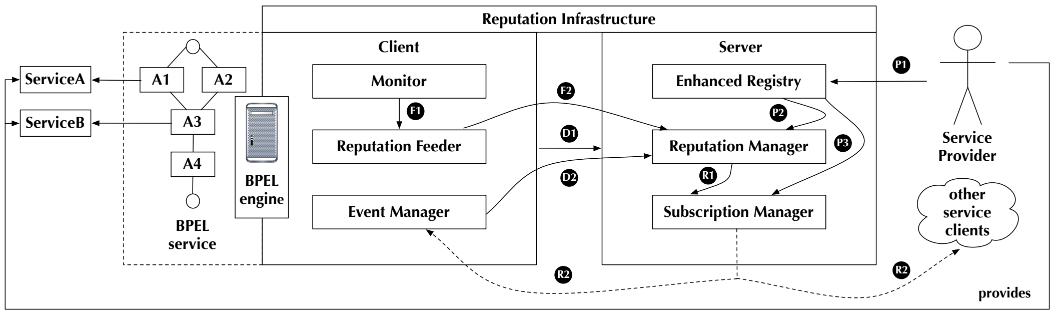


Figure 9.1. **REMAN architecture and system interactions**

9.2 REMAN at a Glance

In the following, we describe the software architecture of REMAN, both at the server- and the client-side. Afterwards, we explain the interactions among the system components.

9.2.1 Server-side Software Architecture

The architecture of REMAN on the server side comprises three main components: the *enhanced registry*, the *reputation manager*, and the *subscription manager*.

Enhanced Registry. It is a UDDI-compliant registry extended with functionalities supporting REMAN. As a UDDI registry, it provides standard UDDI interfaces, which supports service publishing and service discovery. The main extension included by this registry is the functionality to query for QoS estimations of registered services. The registry can be queried by providing either a specific service *TModel* or the concrete service location and the WSDL interface it complies to.

Reputation Manager. It provides functionalities to manage the services registered for reputation and to estimate their QoS. It exposes a public message queue where service clients may post their feedback reports. Moreover, the *Reputation Manager* receives UDDI-related events from the *Enhanced Registry*. For instance, when a new service is registered into the service directory, this component creates the objects needed to represent those entities inside the infrastructure and initializes their reputations to a default value.

The *Reputation Manager* is in charge of managing *reputation policies*. The *reputation policy* is our abstraction for an algorithm that estimates service reputation. Instead of providing an on-line algorithm for immediately processing feedback reports as they arrive, we decided to introduce a scheduler, which invokes reputation policies periodically, to avoid minor fluctuations in the reputation estimates, which could trigger unnecessary notifications.

This component is also aware of the concept of a *reputation era*, which is a fixed-length time interval during which the reputation estimate of resources does not change. All QoS reports received from clients during this period are stored and then processed at the end of the era. This implies that both updates and notifications of reputation-related events happen at the same time, right after the end of an era and before the beginning of the next one. This solution leads to steady QoS estimations through aggregation of feedbacks received within an era.

Subscription Manager. It provides functionalities to notify service consumers when reputation-related events occur and to manage the subscriptions to these events. REMAN supports two event types: *reputation decrease* and *availability of a service with better reputation*.

The former event is fired when the *Reputation Manager* communicates that the reputation of a service has dropped below a certain threshold. Service users are thus notified of a possible failure condition by means of these messages, so that countermeasures can be taken. Upon subscription, each service client specifies the services for which it should receive notifications on reputation decrease and the reputation threshold for each service.

The latter event is used to notify service clients when the set of the "best" services compliant with a particular specification (i.e., the services with the currently-best reputation) changes. By means of these notifications, we let service clients always know which are the best services available on the service market such that when a possible failure occurs they may rebind to another service, which exhibits a better behavior. Service clients subscribe to these events by specifying the WSDL interface they are interested in.

Furthermore, REMAN includes some components implementing side facilities, such as security-related operations (log-in procedures and management of access credentials).

9.2.2 Client-side Architecture

At the client-side, the architecture comprises three components:

Monitor. It monitors the behavior of external services used by the BPEL service client, by checking functional and non-functional properties expressed in WS-CoL [12] and/or ALBERT [9]. Since feedbacks originate from the evaluation of these properties, the latter should specify the interaction with only one service, the one for which the reputation will be computed accordingly.

Reputation Feeder. It provides methods to collect feedback reports and to send them to the server component of REMAN.

Event Manager. It provides functionalities to subscribe to reputation-related events and to react to such notifications.

Figure 9.1 illustrates the components of the architecture, both at the server-side and at the client-side. It also depicts the messages exchanged when interacting with the reputation infrastructure, which are described in the next subsection.

9.2.3 System Interactions

A typical usage scenario of ReMan is the following one:

1. Service providers publish their services (e.g., services A and B) using the UDDI-compliant interface offered by the *Enhanced Registry* (message P1 in figure 9.1). Internally, the *Enhanced Registry* notifies the *Reputation Manager* that a new service has been registered, and thus that a default reputation should be assigned to it (message P2). The *Enhanced Registry* also notifies the *Subscription Manager* such that it can notify interested service clients of the availability of a new service (message P3).

2. When service clients deploy their business processes into the BPEL engine, the client part of ReMan logs into the server part (message D1), in order to get access credentials for subsequent communications. Service clients communicate the selected service bindings to the server using the *Event Manager* (message D2); in this way, clients subscribe to events related to (the type of) services they use. For example, the BPEL service depicted in figure 9.1 will communicate to the *Reputation Manager* its bindings to services A and B, used within the business process by the activities A1 and A3.

3. During execution, each time a client uses an external service, the built-in monitor evaluates a rule associated with the interaction. The result of the evaluation is sent to the *Reputation Feeder* (message F1), which generates a feedback report on the behavior of the external service, to be sent (message F2) to the *Reputation Manager*, on the server component of ReMan.

4. After collecting reputation feedback reports, the *Reputation Manager* updates the reputation estimation of the services registered in the system. Whenever the *Reputation Manager* computes a new value of the reputation of a service, it notifies the *Subscription Manager* (message R1). The latter can then either communicate (message R2) to all subscribed clients that the reputation of a service dropped below a certain threshold, or it can notify them that a new service implementing a certain WSDL interface and with a better reputation became available.

9.3 Reputation Estimation

ReMan has been designed in an open and extensible way, so as to support different methods for computing service reputation and to enable reputation estimation for new kinds of entities (e.g., the reputation of a service provider could be defined by aggregating the feedback reports received for all the services offered by the same provider). The *Reputation Manager* ensures extensibility through the installation of new *reputation policies* provided as plugins.

The default reputation policy plugin in our reference implementation estimates the reputation of each single service published in the infrastructure by using a *binary-based rating approach with reputation propagation*. *Reputation propagation* captures the concept that the *Reputation Manager* builds reputations by using indirect knowledge of services. *Binary-based* means that service clients can rate services by using only boolean values. These values correspond to the evaluation of logical formulae corresponding to the monitored properties, from which feedback reports are created.

This *binary-based rating approach* is based on the *endorsements-refusals ratio* algorithm. It generates the reputation by computing the ratio of the number of positive feedbacks and the total number of feedbacks received until the computation of the estimation is triggered by the system.

Let S be the set of services published in ReMan; F be the set of feedbacks f, where each f is a tuple $\langle s, v, t\rangle$, with $s \in S$ being the service that is the object of the feedback, $v \in \{0, 1\}$ the value of the feedback and $t \in \mathbb{N}^+$ the timestamp at which the feedback is received at the server. Let $F_{s,t} = \{f \in F \mid f.s = s \wedge f.t <= t\}$, $s \in S$, $t \in \mathbb{N}^+$, be the feedback set, i.e, the set of the feedbacks received for a service s until time t; let $P(s,t) = \sum_{f \in F_{s,t}} f.v$ be the amount of endorsement received for service s until time t, and $N(s,t) = |F_{s,t}|$ be the number of total feedbacks received for a service s until time t. The reputation $\rho(s,t)$ for a service s at instant t is then computed using the endorsement-refusals ratio as:

$$\rho(s,t) = \frac{P(s,t)}{N(s,t)}$$

9.4 Implementation

ReMan has been entirely implemented as a JavaEE compliant application; the reason for this choice is that the JavaEE platform is the de facto standard for the development of back-end and distributed applications.

The *Reputation Manager* and the *Subscription Manager* have been implemented by means of both stateless session beans and message-driven ones.

Most of the functionalities of the *Enhanced Registry* have been implemented by means of stateless session beans; standard UDDI services are instead provided by means of Web service beans and by the Grimoires UDDI registry [145]. Grimoires is a registry enhanced with functionalities to add metadata information to UDDI concepts. We adopted it both because it is open-source and because it allows for storing reputation of services directly as a metadata in the registry. Actually, we used a modified version of Grimoires, extended to support notifications about changes in the database of UDDI entities.

Some operations, such as finding the best services compatible with a certain interface, require a notion of service equivalence. We implemented a component that performs the analysis of WSDL documents—associated with UDDI *TModels*—based on syntactic checking[1], and generates sets of compatible services.

At the client-side, service monitoring is performed within the ActiveBPEL engine[2]. The version we have used has been already instrumented with the Dynamo monitoring facility [13]. We extended this version with a functionality to send feedback when a monitoring rule is evaluated, by using an aspect-oriented approach [88] so as to minimize the impact on pre-existing code.

In terms of security, every message exchanged in the system is secured against tampering; access to the system is granted by means of a public key mutual authentication algorithm.

9.5 Experimental Evaluation

We evaluated the performance impact of ReMan with two BPEL processes, the *LoanApproval* process defined in the BPEL specification [5] and the *Radiology* process available in the WSCoL monitoring distribution [13]. In both cases, we implemented the external services required by the BPEL process.

For each process, we measured both the BPEL process deployment time and the process execution time in three different configurations: *(original)* vanilla BPEL engine without any instrumentation; *(monitor)* BPEL engine instrumented to support the Dynamo monitoring facility; *(complete)* BPEL engine instrumented to support both the monitor and ReMan.

For our measurements, all components of our infrastructure as well as the external services required by the two BPEL processes were started on a single machine, an Intel Centrino Duo T2300 CPU with 2GB RAM, running GNU/Linux. To ensure reliable measurements, we removed unnecessary processes as much as possible. ReMan was deployed on JBoss AS 4.2.0-GA. We also used MySQL 5.0.45 as DBMS, ActiveBPEL 2.5

[1]The use of *TModels* (as pointers to WSDL documents) for checking service compatibility is recommended in the UDDI specifications. However, as pointed out by the semantic web research community, a syntactic comparison of WSDL service specifications could not be sufficient for determining service equivalence. Several efforts are dealing with this issue [123, 117]; however they are out of the scope of this work. Therefore, we designed the service equivalence checker as a replaceable component to support different models of service equivalence.

[2]`http://www.activevos.com/`.

as BPEL engine, and the Grimoires Enhanced Registry 1.2.3 as UDDI registry. The BPEL engine and the Grimoires registry were deployed on Apache Tomcat 5.5.25.

Regarding deployment time, we measured the wallclock time taken by the BPEL engine to deploy the process. For each experiment, the application container was restarted. Concerning process execution time, we ran an external client that invoked the BPEL process 10 times with different parameters and we measured the overall wallclock time taken by that client. Due to the complexity of our middleware, measurements are not exactly reproducible; this is a well-known phenomenon, for example in Java-based environments, where measurement variances due to application-inherent non-determinism are often amplified by differences in thread scheduling, dynamic just-in-time compilation, or garbage collection [69]. In order to compensate for the measurement variances, we repeated each experiment ten times (under the same settings) and reported the geometric mean of the ten trials.

As for the test configuration of the reputation infrastructure, we set the reputation era interval to ten seconds, and we used the *endorsements-refusals ratio* as reputation policy.

Tables 9.1 and 9.2 show, respectively, the measured deployment time and the execution time for each process. For each trial, in addition to the execution time, we also show the relative overhead factor (*ovh* column) with respect to the measurement in the original, vanilla setting.

At deployment time, performance degradation is mostly due to ReMan instrumentation. In fact, during this phase, ReMan has to analyze the business process and the WSDL definitions of the external services it uses, to find the remote services within the *Reputation Manager*'s internal registry. The complexity of this phase is proportional to the number of external services the business process interacts with. Indeed, in the case of the *Radiology* process, the analysis takes more time than in the case of the *LoanApproval* process because the former interacts with eight remote services, whereas the latter interacts only with two remote services. This explains why the complete instrumentation of the *Radiology* process causes a deployment overhead of 164% on average, while it results only in 72% overhead for the *LoanApproval* process.

Concerning process execution, the relative overhead due to the instrumentation code is surprisingly uniform for both processes, albeit their execution time in seconds is significantly different. On average, the overhead caused by the monitoring is 26–27%, whereas the overhead for complete instrumentation is about 63%. The relative overhead of the ReMan framework on the top of the monitoring framework is, on average, about 25%.

9.6 Related Work

Several mechanisms to evaluate trust and reputation have been proposed in literature; see [81] for a complete survey and [114] for a classification of such mechanisms driven by the concepts of contextualization and personalization. Although in this chapter QoS refers to the experienced service behavior (including both functional and non-functional aspects), often in the context of SBAs, QoS typically denotes the performance metrics of services [119]. Approaches that select services based on QoS usually extend service registries to support this type of information. An example is UDDIe [133], which extends UDDI with support for the concept of *blue pages*, i.e., information on the QoS properties of a service, which are published by service providers and can be used in queries by service clients. Match-making between properties guaranteed by providers and properties required by clients relies upon the *find* operators defined in the standard UDDI specification. A similar extended registry is described in [141], where a service broker performs QoS-based selection by using an ontology reasoning mechanism for match-making. A common weakness of these approaches is that they rely on the assumption that providers are honest and only advertise QoS properties that they can guarantee.

In [110], the authors propose a conceptual model for Web service reputation; this model is at the basis of an agent-based trust framework for service selection, described in [111]. In contrast to our approach, the authors use a different architectural style, where a software agent is attached to each Web service; the agents are in charge of querying and reporting service reputation. Each service client builds its reputation of services based on the *local* information provided by its neighbors.

A QoS-based service selection model is presented in [103]. The model takes into account the feedback from users as well as other business-related criteria; moreover, it is also extensible, to support multiple QoS selection criteria. Compared to ReMan, it is neither pro-active (because variations of service reputation are not disseminated to other service clients), nor transparent (since service requesters are required to support specific mechanisms for ad-hoc execution monitoring and feedback reporting).

A service recommendation system is proposed in [109]. In this system, clients rate services by using a comparative matrix containing the QoS values advertised by the provider, and the QoS values measured at run time. However, the system does not use Web service standards for service discovery and selection, but relies on ontology-based descriptions. Moreover, user feedback reporting is not automated.

In [140], the authors describe a method to collect monitoring data from clients and to use this information for service recommendations. However, the supported QoS metrics are limited: they support only metrics related to client-side performance, such as throughput, response time, or latency.

A collaborative filtering approach to derive prediction of QoS of Web services that have not been used yet is proposed in [134] and is based on the experience of consumers of similar services. However, the whole approach is poorly integrated in the execution environment and it is neither fully automated nor transparent. Moreover, it supports only the prediction based on the evaluation of timeliness-related QoS properties.

The approach described in [147] adopts a point of view that is complementary to ours. The reputation of a composite service is derived based on the reputation of the single services used within the composition. The reputation mechanism used to compute the reputation of the single services is similar to ours.

The problem of trust and reputation management in open dynamic environments is discussed in [148]. The authors propose some guidelines to build self-organizing referral networks as a means for establishing trust in open environments. However, the technology-agnostic, simulation-based approach adopted in the paper does not allow for a concrete use in Web services-based architectures.

In [35] we proposed an architecture to share reliable service quality information amongst clients, supported by a theoretical model of an incentive-compatible reputation mechanism [83]. However, the requirement of a bank paying for honest feedbacks, postulated by the theoretical model, made the implementation impractical.

9.7 Summary

In this chapter we introduced ReMan, a reputation management infrastructure that supports pro-active service selection for composite Web services. We use monitoring techniques to collect information about functional and non-functional properties of Web service behavior. The resulting feedback data are sent to the server component of our infrastructure, which computes a reputation value for each service registered in the infrastructure. Reputation information is then propagated back to the affected service clients, which can use it to bind to the best available services in the evolving service market.

This feature, integrated in existing state-of-the art run-time infrastructures and compatible with industry standards, fosters dynamic adaptability and self-tuning properties in the execution of composite services.

Table 9.1. **Performance analysis - deployment time**

***(a) LoanApproval* process**

	original [s]	monitoring [s]	overhead	complete [s]	overhead
trial1	3.96	4.12	4.04%	6.91	74.49%
trial2	3.85	4.15	7.79%	6.72	74.55%
trial3	4.03	4.08	1.24%	6.82	69.23%
trial4	3.84	4.13	7.55%	6.74	75.52%
trial5	4.05	4.19	3.46%	6.83	68.64%
trial6	4.01	4.08	1.75%	6.94	73.07%
trial7	3.91	4.08	4.35%	6.72	71.87%
trial8	4.02	4.18	3.98%	6.92	72.14%
trial9	4.01	4.18	4.24%	6.91	72.32%
trial10	3.99	4.11	3.01%	6.87	72.18%
Geometric mean	3.97	4.13	3.63%	6.84	72.37%

***(b) Radiology* process**

	original [s]	monitoring [s]	overhead	complete [s]	overhead
trial1	4.11	4.47	8.76%	11.40	177.37%
trial2	3.97	4.12	3.78%	10.62	167.63%
trial3	4.24	4.43	4.48%	10.75	153.50%
trial4	3.87	4.25	9.82%	10.57	173.04%
trial5	3.95	4.31	9.11%	10.94	176.78%
trial6	4.08	4.38	7.35%	10.91	167.36%
trial7	4.00	4.18	4.50%	10.13	153.17%
trial8	3.80	4.09	7.63%	9.37	146.66%
trial9	3.98	4.24	6.53%	11.22	182.14%
trial10	4.04	4.13	2.23%	9.83	143.49%
Geometric mean	4.00	4.26	5.87%	10.56	163.58%

Table 9.2. **Performance analysis - execution time**

***(a) LoanApproval* process**

	original [s]	monitoring [s]	overhead	complete [s]	overhead
trial1	0.39	0.48	23.08%	0.57	46.15%
trial2	0.35	0.45	28.57%	0.59	68.57%
trial3	0.37	0.52	40.54%	0.65	75.68%
trial4	0.30	0.41	36.67%	0.43	43.33%
trial5	0.45	0.54	20.00%	0.68	51.11%
trial6	0.31	0.38	22.58%	0.57	83.87%
trial7	0.36	0.51	41.67%	0.59	63.89%
trial8	0.36	0.42	16.67%	0.61	69.44%
trial9	0.38	0.44	15.79%	0.65	71.05%
trial10	0.41	0.51	24.39%	0.73	78.05%
Geometric mean	0.37	0.46	25.55%	0.60	63.67%

***(b) Radiology* process**

	original [s]	monitoring [s]	overhead	complete [s]	overhead
trial1	24.78	33.52	35.27%	41.51	67.51%
trial2	21.35	29.75	39.34%	38.65	81.03%
trial3	23.99	28.92	20.55%	37.89	57.94%
trial4	27.63	31.50	14.01%	42.59	54.14%
trial5	22.10	29.73	34.52%	39.82	80.18%
trial6	23.50	34.97	48.81%	37.84	61.02%
trial7	22.84	29.65	29.82%	41.56	81.96%
trial8	27.35	33.24	21.54%	36.92	34.99%
trial9	24.89	28.63	15.03%	40.93	64.44%
trial10	23.21	29.73	28.09%	38.79	67.13%
Geometric mean	24.09	30.90	26.68%	39.61	63.35%

Part V

Finale

Chapter 10

Conclusion

Most traditional software engineering techniques have dealt with systems that lived in a closed, controlled environment. Nevertheless, in the recent years software engineering has shifted towards a type of software that is characterized by a different set of assumptions collectively known as the *open-world assumption*; for this reason, this new kind of software is called *open-world software*.

The open-world assumption is characterized by several facets. Software development and provisioning is decentralized, as it involves multiple stakeholders belonging to different organizations; systems are thus assembled out of components that provide a specific functionality and are provided by independent third parties; bindings among components are often delayed until the execution and may dynamically vary to accommodate changes that support the evolution of the environment with which the system interacts. Finally, the physical deployment of the system requires a heterogeneous and distributed network infrastructure.

Open-world software—such as service-based applications developed by composing different, third-party services—demands for rethinking and extending the traditional software engineering methodologies and the accompanying methods and techniques. In this thesis, we have considered three aspects: *specification*, *verification*, and *reputation management*, and have pursued the following research goal:

> *To design new methods and techniques for specification, verification, and reputation management of open-world software, in particular for the case of service-based applications. These methods and techniques should be i) suitable to deal with aspects such as change, evolution, and reliance on third-parties, and ii) able to improve the overall quality of these applications.*

In the rest of this chapter, we summarize the contributions (section 10.1) of the thesis, and point out its limitations and open issues (section 10.2) as well as future research directions originating from it (section 10.3).

10.1 Contributions

The research goal indicated above has been addressed with the contributions summarized, for each area of interest, in this section.

10.1.1 Specification

Analysis of property specification patterns in SBAs. We run a comparative study on the use of specification patterns in SBAs. We compared the usage of patterns for the requirements specifications of research and industrial case studies, gathered over a time period of more than ten years. The results of this study show that the industrial case studies tend not to use the specification patterns proposed in the research literature, in favor of other patterns that characterize specific aspects of service provisioning and that, conversely, are not common in research case studies.

The SOLOIST specification language. After reasoning on the outcome of the study mentioned above, we designed a new specification language, SOLOIST. Based on a many-sorted first-order metric temporal logic, the language also includes new temporal modalities that have been tailored to express properties that refer to aggregate operations for events occurring in a certain time window. We have also shown how SOLOIST can be translated into linear temporal logic, allowing for its use with established techniques and tools for both design-time and run-time verification.

10.1.2 Verification

Interface decomposition for service compositions. The correct behavior of a service composition, with respect to its requirements specification, depends on a certain, expected behavior of its partner services. However, most of the times the behavioral descriptions of the partner services are unknown. We presented our novel technique to automatically generating the behavioral interfaces of the partner services of a service composition, by decomposing the requirements specification of the composite service.

A syntactic-semantic approach for incremental verification. We introduced a framework, named SiDECAR, for the definition of verification procedures that are automatically enhanced with incrementality by the framework itself. SiDECAR supports a verification procedure encoded as synthesis of semantic attributes associated with a grammar. The attributes are evaluated by traversing the syntax tree that reflects the structure of the software system. By exploiting incremental parsing and attributes evaluation techniques, SiDECAR reduces the complexity of the verification procedure in presence of changes. Hence, it may provide a speed-up of the performance of a verification procedure.

10.1.3 Reputation Management

A pro-active reputation management infrastructure for composite Web services. We presented ReMan, a reputation-aware service execution infrastructure that manages the reputation of Web services used by BPEL orchestrations in an automated and transparent manner. We use monitoring techniques to collect information about functional and non-functional properties of Web service behavior. The resulting feedback data are sent to the server component of our infrastructure, which computes a reputation value for each service registered in the infrastructure. Reputation information is then propagated back to the affected service clients, which can use it to bind to the best available services in the market.

10.2 Limitations and Open Issues

The work presented in this thesis is characterized by various limitations and open issues.

As for the study illustrated in chapter 3, the validity of the results presented in it may be affected by several threats. First, the case studies we have considered from the published research literature may not be adequate representatives of research being developed in the domain of SBAs. Other studies could consider different scientific venues and maybe even extract specifications from case studies presented in different research sub-areas, such as service discovery and dynamic service composition. Similarly, another threat is represented by the fact that we have analyzed the specifications of case studies provided by a single industrial organization. Other industries adopting SOAs could define different requirements for their services. Thus, the results obtained so far could be broadened with a survey involving multiple industrial partners. The matching of specifications with patterns has been performed manually by a single person with six years of experience in the areas of formal specification and verification, as well as service-oriented computing. Other people could classify the specifications differently, especially when the matching with known patterns is not trivial. Furthermore, given that a certain percentage (10% in the case of research literature data, 20% for the industrial ones) of the requirements specifications were expressed using natural language, a certain degree of intrinsic ambiguity is involved in the interpretation of the properties for the purpose of their classification. Finally, there is a inherent limitation in the application of this kind of study, since it only focused on the specification *written* either in papers or in technical documentation. A broader study could have focused on surveying the missing concepts the engineers *wanted* to express, but could not because of, for example, limitations in the specification language.

SOLOIST should not be seen as the “silver-bullet” of specification languages for SBAs. Although it has been designed according to certain predefined requirements (derived from the study reported in chapter 3), it is possible that other contexts or application domains would require new types of operators within the language.

As remarked in section 6.5, the heuristic adopted by our interface decomposition technique may block some good behaviors of the individual services, which instead could be safely allowed. This may happen because an operation of a service that directly leads to the error state, which is the one considered by our heuristic, may be actually triggered by an operation of another service.

Regarding SiDECAR, we have not validated yet the feasibility of one of the requirements for using it: the fact that the verification procedure that one wants to make incremental by means of SiDECAR has to be formalized as synthesis of semantic attributes. This is not a theoretical issue, since it is well-known that attribute grammars have the same expressiveness of Turing machines. Rather, we have not fully assessed the amount of work required to encode the algorithms associated with well-established analysis techniques in an attribute grammar form. However, our preliminary report [24] shows the application of SiDECAR to encode—besides reachability analysis—also reliability prediction of a program, based on the expected reliability of its parts.

As for ReMan, it currently lacks techniques for identifying unfair ratings and thus evaluating raters’ credibility. These techniques could also be the basis for mechanisms to discourage clients from cheating when reporting feedback.

10.3 Future Directions

This thesis sets the basis to follow different research directions in the future.

Concerning SOLOIST, our next steps will focus on its efficient verification based on the Zot toolkit [126], by defining an efficient SMT-based encoding of the language. Although Zot has been used so far for design-time verification, we also want to experiment to embed it and its SOLOIST plug-in within a Web service monitoring architecture (such as Dynamo [13]), to enable support also for run-time verification.

Our approach for decomposing interface specifications can be extended in multiple ways. First, alternative heuristics could assess precisely to which extent a partner service contributes to fulfill (or not) the global requirements, removing the present limitation. This is particularly important in the case in which multiple partner services have operations that could possibly lead to the error state. Secondly, support for the refinement of the generated specifications can be added by extending the analysis of the counterexamples to filter missing behaviors (for example, by performing behavior realizability analysis as suggested in [131]). Last, we will consider the inclusion of the support for timed property specifications, such as those expressed in SOLOIST.

Regarding SiDECAR, the generality of the methodology it advocates will drive us to widen the scope of application to a number of scenarios. For example, at design time, SiDECAR could effectively support designers in evaluating the impact of changes in their products, in activities such as what-if analysis and regression verification, possibly integrated within IDE tools. Existing techniques for automated verification based either on model checking or on deductive approaches, as well as their optimizations, could be adapted to use SiDECAR, exploiting the benefits of incrementality. At run time, the incrementality provided by SiDECAR could be the key factor for efficient online verification of continuously changing situations, which could then trigger and drive the adaptation of self-adaptive systems. Furthermore, SiDECAR could also bring at run time the same analyses so far limited to design time for efficiency reasons.

Future work on SiDECAR will address three main directions. First, we want to support run-time changes of the language (and thus the grammar) in which the artifact to be verified is described, motivated by advanced adaptiveness capability scenarios. Secondly, we want to support specifications that can change, and still exploit the benefit of incremental verification. Last, we will continue our work to develop an incremental verification environment—by incorporating improvements to exploit parallelism [7] and to apply finer incremental parsing techniques—and will conduct experimental studies on real-world applications to quantify the effectiveness of SiDECAR in the definition and the execution of state-of-the-art verification procedures.

As for ReMan, we want to explore methods to make the reputation estimation context-aware such that multiple reputation values can be associated with a service on the basis of the context in which it operates. Moreover, we want to include mechanisms to discourage clients from cheating when reporting feedback.

Finally, our future vision is to integrate our approaches for specification, verification, and reputation management, as well as other approaches [59] developed within our group, into a unified framework that supports continuous quality assurance of open-world software [27].

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language. Towns, buildings, construction*. Oxford University Press, 1977.

[3] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *ICSE 2004: Proceedings of the 26th International Conference on Software Engineering*, pages 168–177. IEEE Computer Society, 2004.

[4] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 265–275. IEEE Computer Society, 2009.

[5] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, 2003.

[6] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the 2006 IEEE International Conference on Web Services*, pages 63–71. IEEE Computer Society, 2006.

[7] A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. PAPAGENO: a parallel parser generator for operator precedence grammars. In *SLE 2012: Proceedings of the 5th International Conference on Software Language Engineering*, 2012. to appear.

[8] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. A timed extension of WSCoL. In *ICWS 2007: Proceedings of the IEEE International Conference on Web Services*, pages 663–670. IEEE, 2007.

[9] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, 2007.

[10] L. Baresi, D. Bianculli, S. Guinea, and P. Spoletini. Keep it small, keep it real: Efficient run-time verification of web service compositions. In *FMOODS/FORTE 2009: Proceedings of IFIP international conference on Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 26–40. Springer, 2009.

[11] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issues and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[12] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC 2005: Proceedings of the 3rd International Conference on Service-Oriented Computing*, volume 3826 of *LNCS*, pages 269–282. Springer, 2005.

[13] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *ESSPE'07: Proceedings of the 2007 International Workshop on Engineering of Software Services for Pervasive Environments*, pages 11–20. ACM, 2007.

[14] D. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In *CAV 2010 : Proceedings of the 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.

[15] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL 2012: Proceedings of the 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 191–202, 2012.

[16] A. Bauer, R. Gore, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *ICTAC 2009: Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, volume 5684 of *LNCS*, pages 96–111. Springer, 2009.

[17] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable Web-services. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 141–150. ACM, 2009.

[18] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *STTT*, 9:505–525, 2007.

[19] D. Bianculli. Lifelong verification of dynamic service compositions. In *FSEDS '08: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium, co-located with ACM SIGSOFT 2008/FSE 16*, pages 1–4. ACM, 2008.

[20] D. Bianculli, W. Binder, L. Drago, and C. Ghezzi. Transparent reputation management for composite Web services. In *ICWS 2008: Proceedings of the IEEE International Conference on Web Services*, pages 621–628. IEEE, 2008.

[21] D. Bianculli, W. Binder, L. Drago, and C. Ghezzi. ReMan: A pro-active reputation management infrastructure for composite Web services. In *ICSE 2009: Proceedings of the 31st International Conference on Software Engineering*, pages 623–626. IEEE, 2009. Formal Research Demo.

[22] D. Bianculli, W. Binder, and M. L. Drago. Automated performance assessment for service-oriented middleware: a case study on BPEL engines. In *WWW 2010: Proceedings of the 19th International Conference on World Wide Web*, pages 141–150. ACM, 2010.

[23] D. Bianculli, W. Binder, and M. L. Drago. SOABench: Performance evaluation of service-oriented middleware made easy. In *ICSE 2010: Proceedings (Volume 2) of the 32nd International Conference on Software Engineering*, pages 301–302. ACM, 2010. Informal Research Demo.

[24] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. A syntactic-semantic approach to incremental verification. Internal Report.

[25] D. Bianculli and C. Ghezzi. Monitoring conversational web services. In *IW-SOSWE'07: Proceedings of the 2nd International Workshop on Service-Oriented Software Engineering*, pages 15–21. ACM, 2007.

[26] D. Bianculli and C. Ghezzi. SAVVY-WS at a glance: supporting verifiable dynamic service compositions. In *ARAMIS 2008: Proceedings of the 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems*, pages 49–56. IEEE, 2008.

[27] D. Bianculli and C. Ghezzi. Towards a methodology for lifelong validation of service compositions. In *SDSOA 2008: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments*, pages 7–12. ACM, 2008.

[28] D. Bianculli, C. Ghezzi, and C. Pautasso. Embedding continuous lifelong verification in service life cycles. In *PESOS 2009: Proceedings of the First International Workshop on Principles of Engineering Service-oriented Systems*, pages 99–102. IEEE, 2009.

[29] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. Specification patterns from research to industry: a case study in service-based applications. In *ICSE 2012: Proceedings of the 34th International Conference on Software Engineering*, pages 968–976. IEEE, 2012.

[30] D. Bianculli, C. Ghezzi, and P. San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *FACS 2012: Proceedings of the 9th International Symposium on Formal Aspects of Component Software*, 2012. To appear.

[31] D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify BPEL4WS workflows. In *SOCA 2007: Proceedings of the 2007 IEEE International Conference on Service-Oriented Computing and Applications*, pages 13–20. IEEE, 2007.

[32] D. Bianculli, C. Ghezzi, P. Spoletini, L. Baresi, and S. Guinea. A guided tour through SAVVY-WS: a methodology for specifying and validating Web service compositions. In *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 131–160. Springer, 2008.

[33] D. Bianculli, D. Giannakopoulou, and C. S. Păsăreanu. Interface decomposition for service compositions. In *ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering*, pages 501–510. ACM, 2011.

[34] D. Bianculli, M. Jazayeri, and M. Pezzè, editors. *Matinée with Carlo Ghezzi - from Programming Languages to Software Engineering*. CreateSpace, June 2012.

[35] D. Bianculli, R. Jurca, W. Binder, C. Ghezzi, and B. Faltings. Automated dynamic maintenance of composite services based on service reputation. In *ICSOC'07: Proceedings of the 5th International Conference on Service-oriented computing*, volume 4749 of *LNCS*, pages 449–455. Springer, 2007.

[36] D. Bianculli, A. Morzenti, M. Pradella, and P. San Pietro and Paola Spoletini. Trio2Promela: a model checker for temporal metric specifications. In *ICSE 2007 Companion: Companion of the proceedings of the 29th International Conference on Software Engineering*, pages 61–62. IEEE, 2007. Informal Research Demo.

[37] D. Bianculli, P. Spoletini, A. Morzenti, M. Pradella, and P. San Pietro. Model checking temporal metric specification with Trio2Promela. In *FSEN 2007: Proceedings of International Symposium on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 388–395. Springer, 2007.

[38] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[39] F. Bitsch. Safety patterns — the key to formal specification of safety requirements. In *SAFECOMP 2001: Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, volume 2187 of *LNCS*, pages 176–189. Springer, 2001.

[40] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, 1(1):75–88, 1984.

[41] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[42] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *LICS'11: Proceedings of the 26th Symposium on Logic in Computer Science*, pages 43–52. IEEE Computer Society, 2011.

[43] I. Brückner. Slicing concurrent real-time system specifications for verification. In *IFM 2007: Proceedings of the 6th International Conference on Integrated Formal Methods*, volume 4591 of *LNCS*, pages 54–74. Springer, 2007.

[44] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 221–230. IEEE, 2004.

[45] M. Chechik and D. O. Paun. Events in property patterns. In *SPIN 1999: Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 154–167. Springer, 1999.

[46] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[47] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS 2003: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[48] C. Conway, K. Namjoshi, D. Dams, and S. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV 2005: Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *LNCS*, pages 387–400. Springer, 2005.

[49] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM, 2000.

[50] S. Crespi Reghizzi and D. Mandrioli. Operator precedence and the visibly push-down property. *J. Comput. Syst. Sci.*, 2011. Accepted for publication.

[51] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA 2006: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24. ACM, 2006.

[52] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, 2005.

[53] L. de Alfaro. Temporal logics for the specification of performance and reliability. In *STACS'97: Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *LNCS*, pages 165–176. Springer, 1997.

[54] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[55] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[56] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *FMSP '98: Proceedings of the 2nd workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.

[57] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420. IEEE Computer Society, 1999.

[58] A. Ershov. On the partial computation principle. *Inf. Process. Lett.*, 6(2):38–41, 1977.

[59] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Asp. Comput.*, 24(2):163–186, 2012.

[60] B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27:253–274, 2005.

[61] M. J. Fischer. Some properties of precedence languages. In *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*, pages 181–190. ACM, 1969.

[62] S. Flake, W. Müller, and J. Ruf. Structured english for model checking specification. In *Trans. Amer. Math. Soc*, pages 2547–2552. VDE Verlag, 2000.

[63] R. W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10:316–333, 1963.

[64] H. Foster. WS-Engineer 2008: A service architecture, behaviour and deployment verification platform. In *ICSOC 2008: Proceedings of the 6th International Conference on Service-Oriented Computing*, volume 5364 of *LNCS*, pages 728–729. Springer, 2008.

[65] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of Web service compositions. In *ASE 2003: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 152–163. IEEE, 2003.

[66] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[67] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[68] A. Gauci, G. J. Pace, and C. Colombo. Statistics and runtime verification. Technical Report 02-WICT-2009, University of Malta, 2009.

[69] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA'07: Proceedings of the 22nd conference on Object-oriented programming systems and applications*, pages 57–76. ACM, 2007.

[70] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, 1979.

[71] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC/FSE-11: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*, pages 257–266. ACM, 2003.

[72] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *ASE 2002: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 3–12. IEEE, 2002.

[73] V. Gruhn and R. Laue. Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[74] L. Grunske. Specification patterns for probabilistic quality properties. In *ICSE 2008: Proceedings of the 30th International Conference on Software Engineering*, pages 31–40. ACM, 2008.

[75] S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC 2008: Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 63–72. IEEE Computer Society, 2008.

[76] S. Hallé, R. Villemaire, and O. Cherkaoui. Specifying and validating data-aware temporal web service properties. *IEEE Trans. Softw. Eng.*, 35(5):669–683, 2009.

[77] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *J. ACM*, 48:880–907, July 2001.

[78] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 180–181. Springer, 2004.

[79] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, 2009.

[80] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[81] A. Josang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decis. Support Syst.*, 43(2):618–644, 2007.

[82] N. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[83] R. Jurca, W. Binder, and B. Faltings. Reliable QoS Monitoring Based on Client Feedback. In *WWW'07: Proceedings of the 16th international conference on World Wide Web*, pages 1003–1011, 2007.

[84] R. Jurca and B. Faltings. Minimum Payments that Reward Honest Reputation Feedback. In *EC'06: Proceedings of the 7th conference on Electronic commerce*, pages 190–199. ACM, 2006.

[85] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini, and M. Jmaiel. Specifying and monitoring temporal properties in web services compositions. In *ECOWS 2009: Proceedings of the 7th European Conference on Web Services*, pages 148–157. IEEE Computer Society, 2009.

[86] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, USA, 1968.

[87] A. Keller and H. Ludwig. The WSLA framework: specifying and monitoring service level agreement for web services. *J. Netw. Syst. Manage.*, 11(1), 2003.

[88] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97: Proceedings of the 11th European conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[89] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2:127–145, 1968.

[90] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 372–381. IEEE Computer Society, 2005.

[91] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.

[92] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, 2004.

[93] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):Article 7, 2007.

[94] I. Krka, Y. Brun, G. Edwards, and N. Medvidović. Synthesizing partial component-level behavior models from system specifications. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 305–314. ACM, 2009.

[95] S. Labbe, J.-P. Gallois, and M. Pouzet. Slicing communicating automata specifications for efficient model reduction. In *ASWEC'07: Proceedings of the 18th Australian Software Engineering Conference*, pages 191–200. IEEE Computer Society, 2007.

[96] F. Laroussinie, A. Meyer, and E. Petonnet. Counting CTL. In *FOSSACS 2010: Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures*, volume 6014 of *LNCS*, pages 206–220, 2010.

[97] F. Laroussinie, A. Meyer, and E. Petonnet. Counting LTL. In *TIME 2010: Proceedings of the 17th International Symposium on Temporal Representation and Reasoning*, pages 51–58. IEEE, 2010.

[98] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE 2008: Proceedings of the 30th International Conference on Software Engineering*, pages 291–300. ACM, 2008.

[99] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[100] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060 – 1076, 1980.

[101] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *FSE SIGSOFT 2004: Proceedings*

of the 12th ACM SIGSOFT international symposium on Foundations of software engineering, pages 53–62. ACM, 2004.

[102] Z. Li, J. Han, and Y. Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC 2005: Proceedings of the 3rd International Conference on Service-oriented computing*, volume 3826 of *LNCS*, pages 73–86. Springer, 2005.

[103] Y. Liu, A. H. Ngu, and L. Z. Zeng. QoS computation and policing in dynamic web service selection. In *WWW Alt. '04: Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters*, pages 66–73. ACM, 2004.

[104] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and participant synthesis. In *WS-FM 2007: Proceedings of the 4th International Workshop on Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 46–60. Springer, 2008.

[105] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 501–510. ACM, 2008.

[106] D. C. Luckham, F. W. von Henke, B. Krieg-Brueckner, and O. Owe. *ANNA: a language for annotating Ada programs*. Springer, 1987.

[107] J. Magee and J. Kramer. *Concurrency: State Models And Java Programs*. John Wiley & Sons, 2nd edition, 2006.

[108] K. Mahbub and G. Spanoudakis. Monitoring WS-Agreements: An event calculus-based approach. In L. Baresi and E. Di Nitto, editors, *Test and Analysis of Web Services*, pages 265–306. Springer, 2007.

[109] U. S. Manikrao and T. V. Prabhakar. Dynamic selection of web services with recommendation system. In *NWESP '05: Proceedings of the International Conference on Next Generation Web Services Practices*, page 117. IEEE Computer Society, 2005.

[110] E. M. Maximilien and M. P. Singh. Conceptual model of web service reputation. *SIGMOD Rec.*, 31(4):36–41, 2002.

[111] E. M. Maximilien and M. P. Singh. Toward autonomic web services trust and selection. In *ICSOC '04: Proceedings of the 2nd International Conference on Service-Oriented Computing*, pages 212–221. ACM, 2004.

[112] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. In *ICFEM'08: Proceedings of the 10th International Conference on*

Formal Engineering Methods, volume 5256 of *LNCS*, pages 105–125. Springer, 2008.

[113] B. Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.

[114] L. Mui. *Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks*. PhD thesis, Massachusetts Institute of Technology, 2003.

[115] C. Müller, O. Martín-Díaz, A. Ruiz-Cortés, M. Resinas, and P. Fernández. Improving temporal-awareness of WS-Agreement. In *ICSOC 2007: Proceedings of the 5th International Conference on Service-Oriented Computing*, volume 4749 of *LNCS*, pages 193–206. Springer, 2007.

[116] S. Murer and B. Bonati. *Managed Evolution: A Strategy for Very Large Information Systems*. Springer, 2010.

[117] M. Nagarajan, K. Verma, A. P. Sheth, J. Miller, and J. Lathem. Semantic Interoperability of Web Services - Challenges and Experiences. In *ICWS’06: Proceedings of the 2006 IEEE International Conference on Web Services*, pages 373–382. IEEE Computer Society, 2006.

[118] OASIS. UDDI Version 2 Specifications. `https://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`, 2002.

[119] L. O’Brien, L. Bass, and P. Merson. Quality attributes and service-oriented architectures. Technical Report CMU/SEI-2005-TN-014, CMU - Software Engineering Institute, 2005.

[120] M. P. Papazoglou. The challenges of service evolution. In *CAiSE 2008: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, volume 5074 of *LNCS*, pages 1–15. Springer, 2008.

[121] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[122] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007.

[123] S. V. Pokraev, D. A. C. Quartel, M. W. A. Steen, and M. U. Reichert. Requirements and method for assessment of service interoperability. In *ICSOC’06: Proceedings of the 4th International Conference on Service-Oriented Computing*, volume 4294 of *LNCS*, pages 1–14. Springer, 2006.

[124] A. Post, I. Menzel, and A. Podelski. Applying restricted English grammar on automotive requirements — does it work? a case study. In *REFQS 2011: Proceedings of the 17th International Working Conference on Requirements Engineering: Foundation for Software Quality*, volume 6606 of *LNCS*, pages 166–180. Springer, 2011.

[125] M. Pradella, A. Morzenti, and P. San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of Software Engineering*, pages 312–320. ACM, 2007.

[126] M. Pradella, A. Morzenti, and P. San Pietro. A metric encoding for bounded model checking. In *FM 2009: Proceedings of the Second World Congress on Formal Methods*, volume 5850 of *LNCS*, pages 741–756. Springer, 2009.

[127] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW 2007: Proceedings of the 16th international conference on World Wide Web*, pages 973–982. ACM, 2007.

[128] A. Rabinovich. Complexity of metric temporal logics with counting and the Pnueli modalities. In *FORMATS 2008: Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, volume 5215 of *LNCS*, pages 93–108. Springer, 2008.

[129] F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service SLAs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 170–180. ACM, 2008.

[130] W. W. Royce. Managing the development of large software systems. In *IEEE WESCON*, pages 1–9. IEEE, 1970.

[131] G. Salaün and T. Bultan. Realizability of choreographies using process algebra encodings. In *IFM 2009: Proceedings of the 7th International Conference on Integrated Formal Methods*, volume 5423 of *LNCS*, pages 167–182. Springer, 2009.

[132] A. Salomaa. *Formal languages*. Academic Press, 1973.

[133] A. ShaikhAli, O. F. Rana, R. Al-Ali, and D. W. Walker. UDDIe: An extended registry for web services. In *SAINT'03: Proceedings of the 2003 Symposium on Applications and the Internet Workshops*, pages 85–89. IEEE Computer Society, 2003.

[134] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei. Personalized QoS Prediction for Web Services via Collaborative Filtering. In *ICWS 2007: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 439–446. IEEE Computer Society, 2007.

[135] J. Simmonds, M. Chechik, and S. Nejati. Property patterns for runtime monitoring of web service conversations. In *RV'08: Proceedings of the 8th International Workshop on Runtime Verification*, 2008.

[136] P. Sistla. Hybrid and incremental model-checking techniques. *ACM Comput. Surv.*, 28(4es), 1996.

[137] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: an approach supporting property elucidation. In *ICSE 2002: Proceedings of the 22rd International Conference on Software Engineering*, pages 11–21. ACM, 2002.

[138] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV 1994: Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *LNCS*, pages 351–363. Springer, 1994.

[139] A. Stefanescu. *Automatic Synthesis of Distributed Systems*. PhD thesis, University of Stuttgart, 2006.

[140] N. Thio and S. Karunasekera. Web service recommendation based on client-side performance estimation. In *ASWEC '07: Proceedings of the 18th Australian Software Engineering Conference*, pages 81–89. IEEE Computer Society, 2007.

[141] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A concept for QoS integration in Web services. In *WQW 2003: Proceedings of the Fourth international conference on Web information systems engineering workshops*. IEEE Computer Society, 2003.

[142] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.*, 35(3):384–406, 2009.

[143] W3C. Web Services Description Language (WSDL) Version 2.0. `http://www.w3.org/TR/wsdl20/`, 2007.

[144] P. Wong and J. Gibbons. Property specifications for workflow modelling. In *IFM 2009: Proceedings of the 7th International Conference on Integrated Formal Methods*, volume 5423 of *LNCS*, pages 56–71. Springer, 2009.

[145] S. C. Wong, V. Tan, W. Fang, S. Miles, and L. Moreau. Grimoires: A Grid Registry with a Metadata-Oriented Interface (part of Cluster Computing and Grid 2005 Works in Progress). *IEEE Distributed Systems Online*, 6(10), 2005.

[146] G. Yang, M. Dwyer, and G. Rothermel. Regression model checking. In *ICSM 2009: Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 115–124. IEEE Computer Society, 2009.

[147] S. J. H. Yang, J. S. F. Hsieh, B. C. W. Lan, and J.-Y. Chung. Composition and evaluation of trustworthy web services. In *BSN '05: Proceedings of the IEEE EEE05 international workshop on Business services networks*, page 5. IEEE, 2005.

[148] P. Yolum and M. Singh. Engineering self-organizing referral networks for trustworthy service selection. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3):396–407, 2005.

[149] J. Yu, T. Manh, J. Han, Y. Jin, Y. Han, and J. Wang. Pattern based property specification and verification for service composition. In *WISE 2006: Proceedings of the 7th International Conference on Web Information Systems Engineering*, volume 4255 of *LNCS*, pages 156–168. Springer, 2006.

Colophon

This document was typeset using pdfLaTeX (included in the MacTeX-2012 distribution), an extension of the LaTeX 2_{ε} typesetting system, originally developed by Leslie Lamport and based on Donald Knuth's TeX.

The body of the text is typeset in 11pt *Bitstream Charter* with math support provided through the `mathdesign` package by the Math Design project. Sans serif text is set in URW Classico (Optima); monospaced text is set in `Bera Mono`.

Text editing was done in *GNU Emacs* using the AUCTeX package. The bibliography and citations were managed using BibTeX with the help of *BibDesk*. Most of the illustrations were drawn using the PGF/Ti*k*Z (`pgf`) package by Till Tantau; plots were graphed with the PGFPLOTS (`pgfplots`) package by Christian Feuersänger. Sources of this document were versioned using the *Apache Subversion* version control system.

The work contributing to this thesis has been written on different models of Apple Macbook Pro (model identifier `MacBookPro2,2`, `MacBookPro6,2`, `MacBookPro10,1`), running, over time, Mac OS X 10.4 Tiger, Mac OS X 10.5 Leopard, Mac OS X 10.6 Snow Leopard, Mac OS X 10.7 Lion, OS X 10.8 Mountain Lion.